



南京大學

研究生毕业论文 (申请硕士学位)

论 文 题 目 SimCity：环境上下文一致性错误

处理与展示平台研究

作 者 姓 名 李晓帆

学 科、专业方向 计算机技术

指 导 教 师 许畅 教授

研 究 方 向 环境上下文处理

2017 年 5 月 23 日

学 号：**MF1433021**

论文答辩日期：**2017 年 5 月 27 日**

指导教师： (签字)

SimCity: An Experimental Platform for Processing Context Inconsistencies and Its Demonstration

by

Xiaofan Li

Supervised by

Professor Chang Xu

A dissertation submitted to
the graduate school of Nanjing University
in partial fulfilment of the requirements for the degree of

MASTER

in

Computer Science and Technology



Department of Computer Science and Technology
Nanjing University

May 23, 2017

南京大学研究生毕业论文中文摘要首页用纸

毕业论文题目：SimCity：环境上下文一致性错误处理与展示平台研究
计算机技术 专业 2014 级硕士生姓名：李晓帆
指导教师（姓名、职称）：许畅 教授

摘 要

我们处于一个普适计算的时代，计算无处不在。海量的智能手机、平板电脑等计算设备会搭载各种传感器采集周围的环境信息。而一类上下文感知应用会利用这些环境上下文信息感知环境的变化，并据此改变自身的行为，为用户提供针对性的服务。然而考虑到环境的复杂性与传感器精度的有限性，采集的上下文中通常会含有噪声。这些不准确的噪声数据会误导应用对环境的正确感知，进而导致应用运行异常。

为了保证应用使用了正确的上下文，可以在应用与传感器采集模块之间加入一个中间件，用来检测并修复不准确的上下文。目前一个常用的做法是先进行约束检测，根据应用周边环境的特点，为该环境中的上下文设计一组一致性约束，每条约束都描述一组上下文需要满足的性质。如果一组上下文违反了某条约束，我们就认为这组上下文中包含一致性错误。下一步就需要采用修复策略对检测到的错误进行修复。鉴于一致性错误处理的重要性，我们设计并实现了一个结合硬件设备与软件系统的环境上下文一致性错误处理与展示平台 SimCity，能够有效地对一致性检测和修复方法进行检验。具体而言本文的主要工作如下：

1. 我们搭建了一个物理实验环境，包含城市建筑、街道、乐高 EV3 程序块、EV3 红外传感器以及蓝牙车辆设备。该迷你城市可以利用众多传感器不断地采集真实的上下文数据，为检验上下文一致性错误处理方法提供真实且充足的数据来源。
2. 我们实现了一个拥有多个交互模块的软件系统，(1) 能自动维护与所有设备的连接，在检测到设备断开后尝试重连；(2) 能自动利用红外传感器采集上下文，并交给作为中间件的一致性检测与修复模块使用；(3) 能通过不断生成任务来持续地操控车辆行为；(4) 能通过分析交通状况，对车辆

进行自适应调度，避免撞车；(5)能根据突发情况及时切换系统的运行状态，执行重置、重定位以及暂停操作，增强系统的鲁棒性。

3. 我们提供了易于理解、操作友好的用户界面，允许用户通过操控车辆、发布任务影响 SimCity 系统的行为。用户可以使用界面启用不同的运行场景，观察对一致性错误的不同处理对系统产生的影响。
4. 我们通过实验考察了 SimCity 的性能。结果表明，SimCity 对一致性错误的正确修复率达到 84%，并且未修复的错误利用 SimCity 的重定位功能进行解决的成功率也在 87% 以上。在检验一类当上下文发生变化即刻进行约束检测的一致性检测方法时，相比构造模拟上下文与记录真实上下文两种常用的检验方法，SimCity 额外发现了一致性错误的误报情况，表现出更好的检验能力。

关键词： 一致性错误检测；一致性错误修复；一致性错误处理检验；普适计算

南京大学研究生毕业论文英文摘要首页用纸

THESIS: SimCity: An Experimental Platform for Processing Context Inconsistencies and Its Demonstration

SPECIALIZATION: Computer Science and Technology

POSTGRADUATE: Xiaofan Li

MENTOR: Professor Chang Xu

Abstract

We are in the era of pervasive computing, or ubiquitous computing. Massive smart phones, tablet PCs and other computing devices equipped with various sensors can sample information about the surrounding environment. One kind of applications called context-aware applications are capable of utilizing contextual information to perceive environmental changes, and adapt their behavior to these changes to provide targeted services for users. However, taking into consideration the complexity of the environment and the limitation of sensor accuracy, the collected contexts usually contain noises. These inaccurate noise data will mislead the application about the accurate perception of the environment, and then cause the application to run abnormally.

To ensure that the application is using the correct contexts, we can add a middleware between the application and the sensor aquisition module to detect and resolve inaccurate contexts. Currently one promising approach is constraint checking. According to the characteristics of the surrounding environment, we design a set of consistency constraints for the sampled contexts. Each constraint describes a property that checked contexts must satisfy. If the checked contexts violates a constraint, we believe these contexts contain an inconsistency. The next step is to select a proper resolution strategy to resolve this detected inconsistency. Due to the importance of inconsistency processing, we designed and implemented an experimental platform named SimCity for processing context inconsistency and its demonstration. It combines hardware and software, and can effectively perform verification on consistency checking and resolution.

The primary contributions of this article are listed as follows:

1. We built a physical experimental platform, including city buildings, streets, Lego EV3 bricks, Lego EV3 infrared sensors and bluetooth device Zenwheels Micro

Cars. This physical mini city utilizes lots of IR sensors to continuously generate real contexts, and provides a real and adequate data source for verifying context inconsistency processing.

2. We implemented a software system with multiple interactive modules. It can (1) automatically maintain the connections with all devices and attempt to reconnect the device after detecting its disconnection; (2) automatically use infrared sensors to generate contexts and submit these to the consistency checking and resolution; (3) constantly control cars by keeping generating tasks; (4) analyze traffic conditions and schedule cars to avoid crashing into each other; (5) switch the system state depending on the sudden incident, and improve system robustness by performing Reset, Relocation and Suspension.
3. We provided an easy-to-use user interface. Users can influence the behavior of SimCity by controlling cars and creating tasks. They can also enable different scenarios to observe the effects of different inconsistency processing.
4. We evaluated the performance of SimCity through several experiments. The results show that the success rate of resolving inconsistencies has reached 84%. For the unresolved inconsistencies, we use Relocation to handle these, and the success rate of Relocation has reached 87%. When verifying one kind of consistency checking which performs constraint checking upon context change, SimCity found a new situation in which the detected inconsistency is a false positive, compared to two verification methods, using mock contexts and recording real contexts. This shows that SimCity has a better ability to verify inconsistency processing.

keywords: consistency checking, consistency resolution, verification of inconsistency processing, pervasive computing

目 录

目 录	v
插图清单	vii
附表清单	ix
1 绪论	1
1.1 研究背景	1
1.2 研究现状	2
1.3 本文工作	3
1.4 本文组织	4
2 相关工作和技术	5
2.1 上下文一致性错误处理	5
2.1.1 一致性错误检测	5
2.1.2 一致性错误修复	8
2.2 一致性错误处理的检验方法	9
3 SimCity 设备安装与部署	11
3.1 设备连接结构	11
3.2 乐高 EV3 程序块	12
3.2.1 规格与接口	12
3.2.2 第三方系统 ev3dev 的刷写与配置	13
3.3 乐高 EV3 红外传感器	16
3.3.1 红外传感器的相互干扰	16
3.3.2 位置部署	18
3.4 蓝牙车辆设备 ZenWheels Micro Car	19
3.4.1 建立蓝牙连接的方案	19
3.4.2 城市街道设置	20

4 SimCity 系统实现	23
4.1 模块层次结构	23
4.2 采样模块	25
4.3 设备连接与控制模块	28
4.3.1 EV3 程序块的连通性检测	28
4.3.2 蓝牙车辆的连通性检测	30
4.4 一致性检测与修复模块	32
4.5 事件监听模块	33
4.6 应用模块	34
4.7 自适应调度模块	35
4.8 状态转移模块	37
4.8.1 初始化状态与正常运行状态	38
4.8.2 重置状态	39
4.8.3 重定位状态	42
4.8.4 暂停状态	45
5 实验与分析	47
5.1 实验设置	47
5.2 用户界面	48
5.3 运行场景演示	53
5.3.1 理想场景 (Ideal Scenario)	53
5.3.2 包含错误的场景 (Noisy Scenario)	54
5.3.3 修复错误的场景 (Fixed Scenario)	56
5.4 实验数据分析	59
6 总结与展望	65
参考文献	67
简历与科研成果	71
致 谢	73
学位论文出版授权书	75

插图清单

2-1 公式 2-1 的一致性计算树	7
3-1 设备连接结构	12
3-2 EV3 程序块正面	13
3-3 EV3 红外传感器	16
3-4 红外传感器与黑色挡板	17
3-5 红外传感器位置部署	18
3-6 Zenwheels Micro Car	19
3-7 城市道路设计图	20
3-8 改造车辆	21
4-1 模块层次结构	24
4-2 采样程序工作流程图	26
4-3 EV3 程序块连通性检测流程图	29
4-4 蓝牙车辆连通性检测流程图	31
4-5 一致性检测与修复流程图	32
4-6 事件监听模块 UML 类图	34
4-7 自适应调度流程图	36
4-8 状态转移模块 UML 状态图	38
4-9 工作线程	40
4-10 重定位示意图	42
4-11 前向重定位递归算法	45
5-1 两类模板	48
5-2 三类一致性约束	49
5-3 启动界面	50
5-4 用户界面	50
5-5 规则界面与统计界面	51

5-6 理想场景运行演示	54
5-6 理想场景运行演示 (续).....	55
5-7 包含错误的场景运行演示	56
5-7 包含错误的场景运行演示 (续)	57
5-8 修复错误的场景运行演示	58

附表清单

5-1 包含错误的场景下的撞车事故	59
5-2 修复错误的场景下的上下文处理	60
5-3 一致性错误修复	60
5-4 约束违反	62
5-5 重定位	63
5-6 打车任务与自适应调度	63

第一章 绪论

1.1 研究背景

近年来，智能手机、平板电脑等移动设备在人们的日常生活中得到了飞速普及。Gartner 提到，在 2016 年，全球用户购买的智能手机的数量达到了 15 亿^①。由于这些智能设备通常都搭载各式各样的传感器，许多利用传感器为人们提供便捷服务的感知应用也随之流行起来。例如，通过 GPS 传感器获取手机的位置信息，地图应用能为行人与车辆进行导航，网购应用可以追踪快递员的行迹，甚至游戏应用可以结合 VR 技术与现实进行互动^②；通过指纹传感器，智能手机可以进行指纹识别，对个人资料进行加密保护。这些应用通过利用传感器采集环境信息来感知周围环境，调节自身的行为，为用户提供针对性服务，被统称为上下文感知应用 (Context-aware application)。

然而实际上，考虑到环境变化的复杂以及传感器精度的有限，传感器在采集上下文数据时会不可避免地产生随机误差，进而导致使用这些数据的上下文感知应用提供错误的服务 [1]。例如，由于 GPS 传感器的定位数据存在误差，地图应用有时会将用户定位到错误的位置，致使导航失败；红外测距传感器也会因为光照强度、物体表面颜色以及光滑程度等多种因素的影响导致测距不准确。已有的研究工作表明，单独检查不准确的上下文可能无法发现问题，但是结合在它前后采集的上下文，就很可能发现它们之间是不一致甚至互相冲突的，且不准确的上下文通常会表现为一个噪点 [2]。基于这些观察，我们就可以通过检测与修复这些上下文中的一致性错误来避免错误的上下文对应用产生不良影响。

鉴于上下文一致性错误的普遍存在以及对应用服务质量的严重影响，我们需要保证一致性错误处理方法的正确与高效，因此如何能正确、直观地对错误处理方法进行检验显得尤为重要。

^①Gartner 公司关于全球智能手机销量的报告. <http://www.gartner.com/newsroom/id/3609817>

^②Pokémon Go. <http://www.pokemongo.com/>

1.2 研究现状

对于上下文中存在的一致性错误，目前常用的做法是采用约束检测 [2, 3]，其思想是根据具体环境的特点，为该环境中的上下文信息设计一系列的一致性约束，每条约束都描述一组上下文之间需要满足的性质。当环境发生变化时，新的上下文被采集，然后放入与它相关的一致性约束中，与其他上下文一起进行检验。只要有一个性质不满足，即只要有一条约束被违反，就表明这个上下文和其余一起检验的上下文违背了环境中的某些特征，它们之间存在不一致性，不能被上下文感知应用直接使用。

下一步，是对检测到的一致性错误进行修复。Xu 等人将众多一致性修复方法大致分为了 4 类 [2]: (1) 丢弃全部 (Drop-all) [4, 5]，丢弃所有不一致的上下文；(2) 丢弃最新 (Drop-latest) [6]，丢弃不一致的上下文中最新的若干条；(3) 丢弃随机 (Drop-random)，随机丢弃不一致的上下文中的一部分；(4) 用户自定义 (User-specified) [7]，让用户做出选择，丢掉某些上下文，或者添加一些缺失的上下文。除了这 4 种修复策略，Xu 等人通过对上下文感知应用以及周围场景的具体分析，也提出了一些启发式方法，以求对不一致的上下文进行最好地修复 [8, 9]。

目前针对上下文一致性检测与修复的研究工作已经有很多。这些工作通常会专注于一个特定的问题，并在这些问题上进行大量实验，以证明研究工作的有效性。这些工作设计实验的一种做法是，设计一个模拟场景并构造大量的模拟上下文，然后对一致性处理方法进行检验 [10–13]。然而模拟上下文可能会由于模拟场景设计得较为简单而具有一定的片面性，无法涵盖现实中可能发生的各种情况；也可能会有些理想化，现实中几乎不可能产生这样的上下文。因此，构造的上下文数据的代表性值得商榷。另一方面，对同一个一致性错误，可以采取多种修复策略进行修复，既可以丢弃可能错误的上下文，也可以对可能缺失的上下文进行补充。虽然这些修复策略都能消除上下文中的一致性错误，但有些策略可能丢弃了正确的上下文或错误地添补了上下文，而把不准确的上下文留了下来。换句话说，某些对一致性错误的修复可能是错误的。由于无法把修复后的上下文再交给应用或场景使用，我们也无从得知应用的反应，因此无法检验修复的正确性。综上，使用构造的模拟上下文来检验一致性处理方法的做法是无法保证有效性的。另一种常用的检验做法是，采集、录制物理环境中产生的真实上下文用以检验 [2, 14]。这在一定程度上减少了模拟上下文

数据的片面性与理想性，但同样的，由于无法将处理后的上下文交给应用使用以获取反馈，我们很难得知这些上下文中是否存在漏报的不一致性错误，或者检测到的一致性错误中是否存在误报，也无法验证修复的正确性。

1.3 本文工作

鉴于检验上下文一致性错误处理的实验容易受到各种对有效性的威胁，本文提供了一种能有效检验与直观展示上下文一致性错误处理的实验平台 SimCity。SimCity 搭建了一个遍布红外传感器的物理迷你城市，通过蓝牙技术远程控制蓝牙小车设备在城市的街道中穿行。SimCity 利用红外传感器不间断地采集真实的环境上下文数据，经过上下文一致性检测与修复，根据修复后的上下文触发各类事件。上层应用会根据监听的事件，对车辆指派任务。SimCity 会使用处理后的上下文分析交通状况，调度车辆在执行任务的过程中避免相撞，从而在物理环境中做出反馈，以有效验证上下文一致性错误处理的正确性。本文的工作有以下几点：

1. 我们搭建了一个物理实验环境，包含城市建筑、城市街道、EV3 程序块、EV3 红外传感器以及蓝牙车辆设备。我们对设备进行了一系列配置与部署，使得该物理环境可以不断采集真实的上下文数据，为上下文一致性检测与修复提供真实、充足的数据来源。
2. 我们实现了一个配套的软件系统，拥有多个相互协作的应用模块，(1) 能周期性检测所有硬件设备的连通性，并在设备断开后尝试自动重连，保障系统正常工作；(2) 能自动采集上下文数据，调用一致性检测与修复方法；(3) 能自动生成任务，不断对车辆发送操控请求；(4) 能智能分析交通状况，对车辆进行安全调度，防止碰撞事故；(5) 能及时切换 SimCity 的运行状态，执行重置、暂停及重定位功能，提高系统的鲁棒性。
3. 我们提供了一个直观、操作友好的用户界面，用户可以通过界面(1) 创建自定义任务，(2) 发送车辆控制请求，(3) 启用不同的运行场景，展示一致性检测与修复的不同效果。
4. 我们仔细设计了实验，通过大量实验数据分析了 SimCity 实验平台的性能。实验结果表明，SimCity 对一致性错误的正确修复率达到 84% 以上，且未修复的错误通过 SimCity 的重定位功能解决的成功率也达到 87%。相比通常的构造模拟上下文与记录真实上下文的检验方法，SimCity 结合物

理环境与用户界面，针对具有代表性的 Pcc 检测方法 [2] 额外发现了一致性错误的误报情况，表现出更好的检验效果。

1.4 本文组织

本文剩余章节的组织安排如下：第二章介绍与本文工作相关的工作与技术，包括一致性错误的检测与修复以及通常的检验方法；第三章会讲解 SimCity 所有硬件设备的安装与配置工作；第四章详细介绍 SimCity 软件系统的整体框架以及各个模块的实现细节；第五章会分别演示 SimCity 的三种运行场景，并通过分析实验数据展示 SimCity 的性能；第六章是对本文工作的一个总结。

第二章 相关工作和技术

2.1 上下文一致性错误处理

2.1.1 一致性错误检测

在普适计算环境中，计算设备会通过多种传感器不断地采集周围的环境信息，如位置、速度、温度等。这些环境信息被归类为上下文 [15, 16]，而根据环境上下文来调整自身的行为，为用户提供针对性服务的一类应用被称为上下文感知应用。伴随着智能手机与平板电脑等移动计算设备的迅速普及，许多上下文感知应用也涌现出来，并被大众广为接受。然而目前的一个现实是，采集的上下文中经常含有噪声，这些不准确或是错误的上下文会严重地影响应用的行为，为用户提供错误的信息与指示。产生噪声数据的原因多种多样。复杂多变的环境本身对传感器提出了很高要求，而传感器精度也是有限的，测量数值也会由于环境条件、操作人员以及传感器自身的不稳定而产生随机误差。众多因素的叠加使得存在上下文噪声的现象非常普遍。

前面提到，上下文噪声会干扰应用的正常运行，因此需要被检测与修复后才能被应用使用。目前常用的一种做法是约束检测，根据具体应用场景的特点，抽象出一组上下文所必须保持的性质 [17]，这些性质被称为一致性约束 [18, 19]。如果有一组上下文违反了某条一致性约束，我们就认为它们是不一致的，它们含有一致性错误，需要被进一步修复。在本文中，和许多已有的研究工作 [15, 20–22] 类似，我们使用一阶逻辑语言 (First-order logic, FOL) 来描述一致性约束，它的语法如下：

$$\begin{aligned} f ::= & \forall \gamma \in S(f) \mid \exists \gamma \in S(f) \mid (f) \wedge (f) \mid (f) \vee (f) \mid \\ & (f) \rightarrow (f) \mid \neg(f) \mid bfunc(\gamma_1.p_1, \dots, \gamma_n.p_m). \end{aligned}$$

在这里， S 表示一个上下文的有穷集合，例如 5 条关于车辆 A 最近的位置信息的集合。变量 γ 表示上下文集合中的任意一个上下文实例，例如车辆 A 最近的一条位置信息。一条上下文可以拥有多个域，例如主体名称、时间、动作、

地点等，而一条上下文实例就必须在所有域上都具有明确的取值。每个上下文集合都对应一个模板 (Pattern)，对上下文的某几个域的取值进行限制，只有满足了这些限制的上下文实例才可以加入这个集合中。例如，对前面举例的上下文集合，可以限制域主体名称为车辆 A 。 $bfunc$ 代表一个用户自定义的布尔函数，它以上下文实例中域的取值作为参数，返回 $True$ 或者 $False$ 。例如，可以定义一个函数 $diff(\gamma_1.subject, \gamma_2.subject)$ ，判断两条上下文实例是否具有不同的主体。

下面举例来说明一致性约束的建立与使用。设想一个简单的应用场景：车辆在城市中不断穿行，车载 GPS 会不断获取车辆的位置信息。我们假定 GPS 传感器采集的上下文只有三个域：车辆名称 $name$ 、车辆位置 loc 以及采集时间戳 $timestamp$ 。首先，我们可以建立只限制车辆名称的模板，将 $name$ 取值指定为车辆 A ，这样就把所有与车辆 A 有关的上下文实例都加入到同一个集合中 (命名为 $CarA$)。考虑到时间与物理空间的约束，我们可以建立一条一致性约束：车辆 A 在同一时间内不可能位于两个不同地点。这条约束使用一阶逻辑语言可以表达成下面的形式：

$$\begin{aligned} & \forall \gamma_1 \in CarA (\neg \exists \gamma_2 \in CarA (\\ & \quad \neg sameLoc(\gamma_1.loc, \gamma_2.loc) \wedge sameTime(\gamma_1.timestamp, \gamma_2.timestamp))). \end{aligned} \quad (2-1)$$

这条一致性约束从上下文集合 $CarA$ 中任意取出两条上下文实例 γ_1 与 γ_2 ，同时对域 loc 与 $timestamp$ 进行比较。假设车辆 A 真的同时出现在两个不同地方，那么集合 $CarA$ 中必然有两条上下文实例在 $timestamp$ 上取值相同，而在 loc 上取值不同。当 γ_1 与 γ_2 分别赋值为这两条实例时，一致性约束计算的真值为 $False$ ，表明该约束被违反，这两条上下文实例之间存在一致性错误。

下面以具有代表性的 Pcc (Partial constraint checking) 检测方法 [2] 为例解释约束检测的计算过程。Pcc 对每条一致性约束都建立一棵一致性计算树 (Consistency computation tree, CCT)，用以存储计算出的真值与选择的上下文实例，以备后续计算使用。CCT 的非叶子节点代表一个量词或其他的逻辑算子，叶子结点代表一个 $bfunc$ 。以公式 2-1 为例，假设上下文集合 $CarA$ 包含两个上下文实例 ($CarA = \{ctx_1, ctx_2\}$)，那么它对应的计算树如图 2-1 所示。从计算树中可以看出，量词节点 $\forall/\exists \gamma \in S$ 会为上下文集合 S 中的每一个上下文实例 ctx_n 都建立一个分支，并给每个分支都建立一棵相同的子计算树。在量词节点的不同的分支上，变量 γ 与不同的上下文实例绑定，这种绑定关系向下传递到

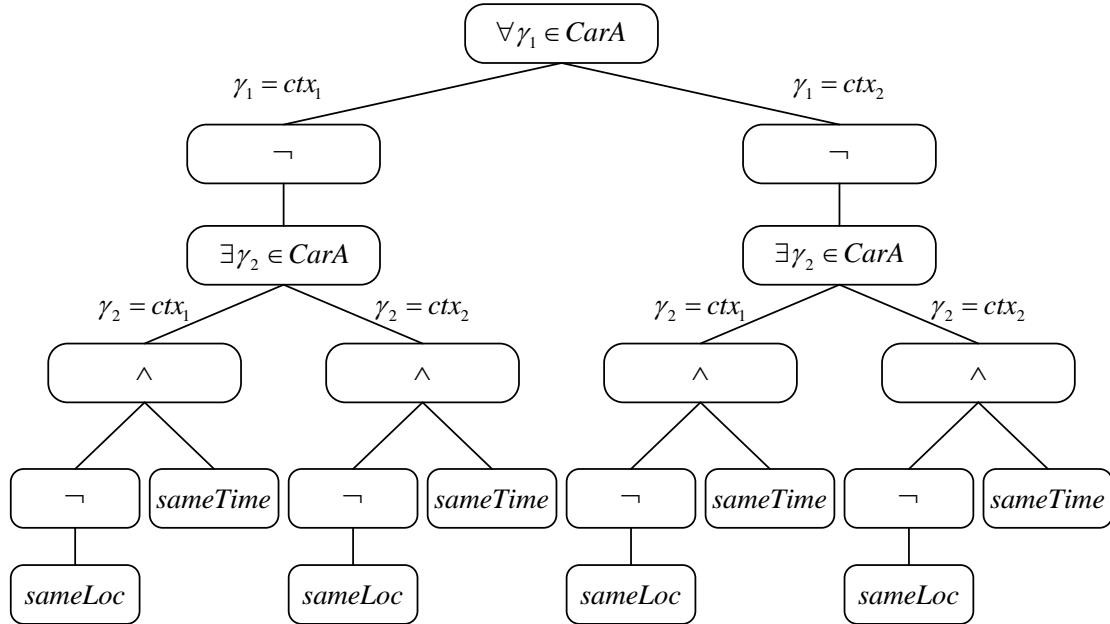


图 2-1: 公式 2-1 的一致性计算树

子计算树中。到了叶子节点，根据变量的绑定关系，我们可以计算出 $bfunc$ 的真值，然后自底向上推算出所有节点的真值。而根节点的真值就代表了这整条一致性约束是否被违反。每个节点计算出的真值都会存储在该节点上，当某个量词节点的上下文集合发生上下文变化时，(1) 对于新增的上下文实例，在该节点下添加一个新的分支，将量词中的变量与该实例绑定，并构建一个相同的子计算树，然后只需重新计算从该计算树的叶子结点到根节点的真值；(2) 对于删掉的上下文实例，直接删除量词节点下的对应分支，并重新计算从量词节点到根节点的真值。至于如何复用子节点的真值计算结果来计算新的真值，[2] 中列出了详细的规则，这里不再赘述。

紧随真值计算的下一步，是获得引起一致性错误的上下文实例与变量之间的绑定关系，也就是违反一致性约束的一组上下文。Pcc 借鉴了 [23] 中提出的一致性链 (Link) 的概念来表达这种绑定。Pcc 定义了两种链，满足链 (Satisfaction link) 与违反链 (Violation link) 分别用来储存满足或违反某个一致性约束的一组绑定关系。一致性链的形式为 $(type, bindings)$ ，其中 $type$ 取值为 *satisfied* 时表示这是满足链， $type$ 取值为 *violated* 时则表示这是违反链。 $bindings$ 是量词变量与上下文实例之间绑定关系的集合。图 2-1 中最左边的 $bindings$ 可以表示为 $\{(\gamma_1, ctx_1), (\gamma_2, ctx_1)\}$ 。与真值计算类似，一致性链的生成也是自底向上进行的。复用旧一致性链生成新链的规则在 [2] 中可以查看，这

里同样不再赘述。

为提高约束检测的效率，除了像 Pcc 这样用空间换时间，复用已有的计算结果来加快计算，还可以充分利用 CPU 或者 GPU 的多核优势，将串行的计算过程并行化 [14, 24]。Sui 等人 [14] 提出了一个并行化约束检测的方法 GAIN，能够将约束检测中的计算任务分发给多个 GPU 线程去执行。另外，约束检测时机的不同也可能会造成不同的检测效果。由于延迟等因素的存在，有些一致性错误可能会随着迟到的上下文的来临而消失。SHAP [1, 12] 和 Cina [25] 这两份工作都对这类情况进行了讨论与处理。

2.1.2 一致性错误修复

检测到一致性错误，并得到违反一致性约束的上下文后，就需要对这些上下文实例进行修复以消除不一致。在软件工程领域，针对一致性错误的修复已有了许多研究工作。之前提到的 Xlinkit [23, 26] 是针对 XML 文档中存在的一致性错误进行修复，但它的修复过程需要人工干预，是根据预定义的规则向用户提示一些修复选择。Egyed 等人的工作 [27] 是修复 UML 模型中存在的一致性错误，而他们修复错误的过程也需要人工的介入。上面这些工作的研究对象是 XML 文档与 UML 模型，它们有一个共同的特点，环境相对固定，上下文变化不太频繁，因此采用人工干预的修复方式的代价可以接受。但是在普适计算领域，面对瞬息万变的环境和频繁产生的上下文，人工交互效率低下的缺点会被无限放大，这种修复方式根本来不及处理高频产生的一致性错误。因此，在普适计算环境中需要人工介入的一致性修复方式是不可行的。另外还有一些研究是专注于修复移动应用内部或者移动应用之间的策略冲突 [7, 28]，但是这些工作都假设应用能够接收到准确的上下文，并且知道如何对策略冲突做出正确的修复。而在普适计算领域，这些假设都不成立，因此这些修复方法也无法应用到普适计算中。

Xu 等人将已有的一致性修复策略进行了大致分类 [11]。对于一组含有一致性错误的上下文，为了消除该错误，可以简单地丢掉所有的上下文实例 (Drop-all) [4, 5]，也可以丢掉部分最新的上下文实例 (Drop-latest) [6]，或者随机地丢弃一些上下文实例 (Drop-random)，或者让用户参与到修复过程中 (User-specified) [7]。此外，Xu 等人也使用了一些启发式方法，通过对应用本身以及使用场景的分析，丢掉那些不准确的可能性最高的上下文实例 (Drop-bad) [9]。还有一些研究工作采用了混合型修复策略 [29]，将低层的一致性错误修复与高

层的应用错误恢复结合在一起。这种策略基于上下文感知应用的语义，提出了一个上下文关联模型来计算上下文出错的可能性，又提出了一个成本模型来平衡一致性错误修复的正确率与应用错误恢复之间的开销。

2.2 一致性错误处理的检验方法

目前研究普适环境中的上下文一致性错误的研究工作有很多，且他们通常会专注于一致性检测或者修复方法的一个方面。例如，[3] 专注于细化一致性约束的优化粒度，提高约束检测的效率；[14] 专注于利用 GPU 并行化约束检测的计算过程；[12] 专注于运用模式学习来抑制一类称为“上下文一致性错误冒险”的误报。这些工作通常会设计大量实验来检验其有效性，下面我们会列出一些常用的检验方法进行说明。

陈等人提出了一种自动修复不一致上下文的技术 [10]，能够根据用户需要生成抽象修复用例，再通过自动执行用例进行修复。这份工作在实验中模拟了一个拥有两个车间与三条传送带的工厂。每个产品上都有一个 RFID 标签，而在传送带两段各有一个 RFID 阅读器用以读取标签。在读取标签时，就可能发生漏读或者错读等不一致性问题。实验使用模拟的上下文检验修复的成功率时，仅仅是通过执行修复用例后观察模拟上下文是否恢复一致来进行判断，而没有将修复后的上下文投入模拟场景中使用以进一步验证修复的正确性。由于没有将成功修复的正确率考虑在内，该技术的有效性值得商榷。

Xu 等人研究了一致性错误修复方法对应用引起的副作用 [11]。实验使用一个开源的上下文模拟器 Siafu^①，模拟了德国小镇 Leimen 的市民活动。实验比较了不同修复策略对同样的上下文的修复结果，衡量标准有上下文的数量、类型等。然而，由于无法将模拟的上下文投入模拟场景中使用，实验并没有将修复正确率作为一个重要的衡量标准纳入考量。

Xi 等人提出了一种针对一致性错误冒险的抑制技术 [12, 13]，可以用来检测一种特殊的上下文错误。实验共使用了三组上下文数据对该技术进行检验。实验分别模拟了一个智能展厅系统和一个智能照明系统用以采集模拟数据，另外还使用了一组由 UbiComp 2014^② 提供的真实的智能手机使用数据。虽然这份工作专注于一致性错误的检测而不考虑修复，但是是否存在漏报的错误以及检测到的错误中有没有误报，这些问题还是需要结合修复方法将修复后的上下文投

^①Siafu. <http://siafusimulator.org/>

^②UbiComp 2014. <http://ubicomp.org/ubicomp2014/calls/competition.php>

入实验环境中使用才能得到真正结果。

Xu 等人 [2] 与 Sui 等人 [14] 分别在不同方面对约束检测的效率进行了改进。在设计实验时，他们均使用了 SUTPC 系统 [2] 所采集的某城市在 24 小时内的真实出租车数据。同样，虽然两份工作只关注一致性错误的检测，但由于实验脱离了真实的应用环境，他们无法发现一致性错误的漏报与误报情况。

对于上面关于一致性错误处理的研究工作中采用的检验方法，我们可以分为两类：构造模拟上下文与记录真实上下文。模拟上下文采集自模拟场景中，由于场景的不完善以及环境因素的简单化处理，这些模拟上下文既可能具有片面性，无法涵盖现实中的各种复杂情况，也可能具有理想性，根本不会出现在物理环境中。换句话说，模拟场景的设计会直接影响模拟上下文数据的代表性。而由于我们难以度量一个模拟场景的真实性，因此也就很难证实模拟上下文的代表性。第二类检验方法是记录真实场景或应用产生的上下文用作一致性错误处理。这种做法随着记录数据量的增加会削弱片面性和理想性的影响，但是检验方法脱离了真实的应用环境，无法将处理后的上下文重新投入环境中获得反馈。对于一致性检测来说，难以得知是否存在漏报的一致性错误，以及检测出的一致性错误中有没有误报；对于一致性修复来说，修复的正确性无法得到验证。这些脱离应用环境进行检验的有效性威胁，对于构造模拟上下文的检验方法也同样适用。综上，这两种检验方法都不能很好地对一致性错误处理进行有效检验。

第三章 SimCity 设备安装与部署

在本章中，我们会详细介绍 SimCity 实验平台所使用的硬件设备，包括相关的设备功能，设备的连接与安装，嵌入式系统的写入与配置，设备部署中遇到的问题与解决方法等。

3.1 设备连接结构

SimCity 是一个包括软件系统与硬件设备的实验平台。对于硬件设备，我们的要求是能为软件系统源源不断地提供上下文，并且系统可以操控硬件设备做出反应，从而产生新的上下文。结合以上要求，我们设计并搭建出一个物理的迷你城市：多条道路纵横交错，各式建筑矗立其间，道路旁密集地布设着红外传感器，系统遥控车辆在道路上穿行，在经过传感器时就会产生上下文，而根据上下文，系统能够分析交通状况，对车辆进行自适应调度，又影响新的上下文产生。

图 3-1 是 SimCity 各硬件设备互连的结构图。软件系统的控制中心运行在计算机上，采样程序则运行在乐高 EV3 程序块 (Lego EV3 Brick) 上。EV3 程序块通过乐高数据缆线与 EV3 红外传感器相连，不断采集上下文数据。程序块配备了无线网卡，与计算机处于同一局域网中。每当采集到新的传感数据，EV3 程序块上的采样程序就通过无线网络将数据发送给计算机上的控制中心。控制中心对上下文进行处理与分析后，利用蓝牙技术远程控制名为 ZenWheels Micro Car^①的车辆正常行驶或紧急避障。在目前搭建的迷你城市中，共计 32 个 EV3 红外传感器分别连接在 10 个乐高 EV3 程序块上，同时进行采样工作。10 个程序块之间相互独立，均使用 UDP 协议将最新的传感数据发送给计算机上唯一的控制中心。控制中心根据用户选择，可以同时连接与控制多辆小车。这里需要说明一下，采样程序选用 UDP 而不是 TCP 向控制中心发送数据包的原因有：
(1) 考虑到计算机和 EV3 程序块处于一个规模很小的局域网里，丢包率几乎为 0，此时 UDP 的可靠性完全可以满足 SimCity 的需求；(2) 使用 UDP 使控制中

^①ZenWheels Micro Car. <http://zenwheels.com/>

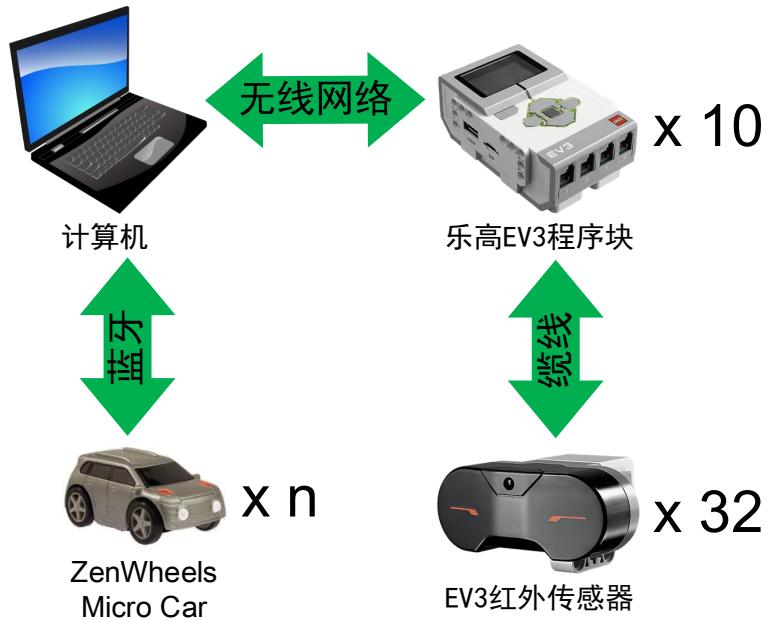


图 3-1: 设备连接结构

心不必维护多个与 EV3 程序块的连接，降低了系统资源开销。

3.2 乐高 EV3 程序块

3.2.1 规格与接口

乐高 EV3 程序块是一个搭载 ARM9 处理器，拥有多个输入输出接口，能自由进行功能扩展的嵌入式系统。EV3 程序块正面如图 3-2，上方为一块 178×128 分辨率的黑白显示屏，用以显示程序块系统的菜单选项以及运行程序的输出信息。返回键可用来撤销操作、终止运行程序及关闭 EV3 程序块。中键相当于 PC 键盘的 Enter 键，用以确认各项输入。上、下、左、右用来导航显示屏中选中的选项。在 EV3 程序块左侧有一个 USB 2.0 接口，我们插入了一个无线网卡以连接到无线网络中，与计算机建立通信。USB 接口旁有一个微型 SD 卡槽，我们将第三方 Linux 系统写入微型 SD 卡并插入卡槽，让程序块通过读取微型 SD 卡来加载启动写入的 Linux 系统。在程序块底部共有 4 个编号为 1、2、3、4 的输入端口，使用乐高数据线可以将最多 4 个 EV3 红外传感器连接至程序块。



图 3-2: EV3 程序块正面

3.2.2 第三方系统 ev3dev 的刷写与配置

乐高 EV3 程序块的官方固件是一个精简的 Linux 系统。为了培养机器人爱好者与编程爱好者的兴趣，官方系统在许多方面进行了简化，这对我们开发程序块的采样功能产生了诸多限制，具体有：(1) 系统启动时无法自动打开无线网络功能并连接到已记录的网络中；(2) 程序需要通过 USB 连接线从 PC 传输到程序块上；(3) 官方仅提供了图形编程模式，无法满足 SimCity 的采样要求。有鉴于此，我们需要给程序块写入一个功能完善，能支持主流编程语言的操作系统。目前，我们选用了 ev3dev^①，一个兼容乐高 EV3 程序块的完整 Debian 发行版。ev3dev 支持 Python、Java、C、C++ 等众多编程语言，且内置了 SSH、APT、Vim、NFS 等诸多工具。简单来说，ev3dev 基本具有一个 Linux 发行版的所有功能，可以满足我们的采样需求。下面是写入 ev3dev 的步骤：

1. 将微型 SD 卡插入读卡器，再将读卡器插入 PC。
2. 安装镜像烧录软件 Etcher^②。
3. 从 ev3dev 的 GitHub 仓库^③下载系统镜像。
4. 使用 Etcher 将 ev3dev 镜像写入微型 SD 卡。
5. 将微型 SD 卡插入 EV3 程序块的 SD 插槽，按下中键启动 ev3dev 系统。

写入和启动 ev3dev 系统的过程不会删除 EV3 官方固件。拔出微型 SD 卡后，启动程序块会再次加载官方系统。

^①ev3dev. <http://www.ev3dev.org/>

^②Etcher. <https://etcher.io/>

^③ev3dev Github. <https://github.com/ev3dev/ev3dev>

下一步需要对 ev3dev 系统进行一系列的配置工作。在 SimCity 系统中，PC 端控制中心会通过 SSH 远程登录 ev3dev 系统，执行采样脚本、关闭或者重启 EV3 程序块。为避免每次执行 SSH 远程命令都要输入 ev3dev 系统的用户密码，我们需要配置从 PC 到 ev3dev 系统的 SSH 免密码登录：

1. 在 PC 上打开一个终端，输入命令：

```
ssh-keygen -t rsa
```

所有选项均输入回车采取默认值，在 PC 的当前用户目录%PC_User% 下将生成.ssh 文件夹以及其中的私钥文件 id_rsa 与公钥 id_rsa.pub。

2. 在终端输入命令：

```
ssh-keyscan $ev3dev_IP >> %SimCity_Dir%/runtime/known_hosts
```

将 ev3dev 的 IP 地址与对应公钥存储到 SimCity 项目中。这里\$ev3dev_IP 表示 ev3dev 系统的 IP 地址，%SimCity_Dir% 表示 SimCity 项目的目录。

3. 在终端输入命令：

```
ssh-copy-id -i ~/.ssh/id_rsa.pub robot@$ev3dev_IP
```

将 PC 端 SSH 公钥添加到 ev3dev 的/home/robot/.ssh/authorized_keys 中。这样 PC 就能够以 robot 用户身份通过 SSH 免密码登录 ev3dev 系统来执行远程命令。

4. 在终端输入命令：

```
ssh-copy-id -i %SimCity_Dir%/runtime/id_rsa.pub robot@$ev3dev_IP
```

将 SimCity 项目中的公钥也添加到 robot 用户的已验证公钥中，这样 PC 端控制中心也能以 robot 用户通过 SSH 免密码登录 ev3dev 系统，以远程启动 ev3dev 上的采样程序。

5. 在终端输入命令：

```
ssh robot@$ev3dev_IP
```

免密码登录 ev3dev。首次连接 ev3dev 时会提示需要验证主机的真实性，输入 yes 自动将 ev3dev 的公钥储存到%PC_User%/.ssh/known_hosts 文件中。

6. 以用户 robot 成功登录 ev3dev 后，输入命令：

```
sudo su
```

切换至 root 用户后，再输入命令：

```
mkdir ~/.ssh
```

在 ev3dev 系统上创建文件夹/root/.ssh。

7. 以 root 用户输入命令：

```
cp /home/robot/.ssh/authorized_keys ~/.ssh/
```

将 robot 的已验证公钥拷贝给 root 用户。这样 PC 端就能以 root 用户用 SSH 免密码登录 ev3dev 并执行远程命令。

我们配置了 PC 端两组 SSH 公私钥，一组位于%SimCity%/runtime/，用于 SimCity 在 PC 端的控制中心启动 ev3dev 上的采样程序与关闭或重启 EV3 程序块；另一组位于%PC_User%/.ssh/，用于 PC 用户 SSH 登录 ev3dev，目的是方便用户批量管理 EV3 程序块，包括批量远程关闭或重启，批量传输文件等。

下一步需要将采样程序 sample.py、启动脚本 start.sh 以及停止脚本 stop.sh 一起拷贝到 ev3dev 上。启动脚本在运行采样程序前会先检测是否已经有采样程序在运行，保证永远只会运行一个。由于设置了 SSH 免密码登录，我们可以直接使用 SCP 命令进行文件传输。首先还是在 PC 上打开一个终端，依次输入命令：

```
cd %SimCity_Dir%/brick
scp sample.py start.sh stop.sh root@$ev3dev_IP:~/
```

采样程序会由 SimCity 的 PC 端控制中心自动运行，不需要也不应当由用户手动运行。

SimCity 对 10 个 EV3 程序块依次编号为 0 到 9，我们需要将 ev3dev 系统的 hostname 设置为它对应的编号。原因有两点：(1) 不同程序块上连接的红外传感器数量不同，采样程序需要根据 ev3dev 的 hostname 判断可以使用的传感器个数，避免抛出异常；(2) 采样程序需要利用 hostname 为发送给 PC 的传感数据标记来源。下面为设置 ev3dev hostname 的步骤：

1. 在 PC 上打开一个终端，输入命令：

```
ssh root@$ev3dev_IP
```

2. 以 root 登录 ev3dev 后，输入命令：

```
echo $brick_id > /etc/hostname
```

设置 hostname 为该程序块的编号。这里\$brick_id 表示程序块的编号。



图 3-3: EV3 红外传感器

3.3 乐高 EV3 红外传感器

乐高 EV3 红外传感器如图 3-3 所示，它使用一根乐高数据线连接到 EV3 程序块的其中一个输入端口中。EV3 传感器有三种工作模式：(1) 近程模式，用于测量物体与传感器之间的距离；(2) 信标模式，追踪配套的红外信标设备；(3) 远程模式，识别远程红外信标发射的信号，遥控程序块。SimCity 仅用到了近程模式进行测距，传感器的工作方式是用上方中央的红外发射器发射出红外线，再用两边的红外接收器接收从物体表面反射回来的红外线，根据往返时间计算物体与传感器之间的距离。但实际上，直接计算极短的往返时间非常困难，通常是利用红外线往返产生的相位移来推算得到的。这个传感器使用 0(很近) 到 100(很远) 之间的数值来表达距离，该数值没有计量单位，仅表示长度的相对大小。在实际工作过程中，通常当物体距离传感器 5 厘米以内时，传感器只会报告 0 值；而当物体距离传感器过远（通常 70 厘米以上）或者传感器测距失败时，传感器会传回 32 位有符号数的最大值 2147483647。

3.3.1 红外传感器的相互干扰

由于 EV3 红外传感器在测距时既会发射红外线，也会接收红外线，因此如果有多个红外传感器在同时工作，一个传感器发射的红外线可能被另一传感器接收到，造成该传感器测距不准确甚至失败。在搭建的迷你城市中，32 个红外传感器部署在尺寸约为 $100\text{cm} \times 90\text{cm}$ 的城市各处，所有传感器朝向各异，且都处于同一水平面内。因此如果不采取一定防范措施，32 个红外传感器同时工作时相互干扰的现象将会十分常见，其测距功能也会受到严重影响，最终导致 SimCity 无法正常工作。

另外我们在前面提到，当物体距离传感器过近时（通常 5 厘米以内），传感器只会报告 0 值。在迷你城市中，传感器为了检测蓝牙车辆经过，是紧贴着



图 3-4: 红外传感器与黑色挡板

路沿布设的。显然当车辆驶过传感器时，两者间距不足 1 厘米，传感器此时会返回 0 值。然而当车辆离开后，有些传感器由于正对着其他传感器或者建筑并且距离较近，也会返回 0 或者很小的数值。这些传感器由于车辆经过前后的读数变化不明显，无法有效区分车辆是否经过，也会影响 SimCity 正常运行。

为解决红外传感器相互干扰以及部分传感器在车辆经过前后读数变化小的问题，我们在迷你城市中，每个红外传感器的正对面，都树立起一块表面粗糙的黑色挡板，如图 3-4 所示。挡板能防止其他传感器发射的红外线对挡板后的传感器造成干扰，同时粗糙的黑色表面能够大量吸收这个传感器自身发射的红外线，降低反射回来被接收的红外线强度，从而增大相位偏差，延长推算出的往返时间，明显提高传感器的测量读数。经过实践，这些挡板立在红外传感器正前方 5 厘米处，能让传感器的测量读数从 0~8 提高到 14 以上，使得传感器能有效感知到小车经过。

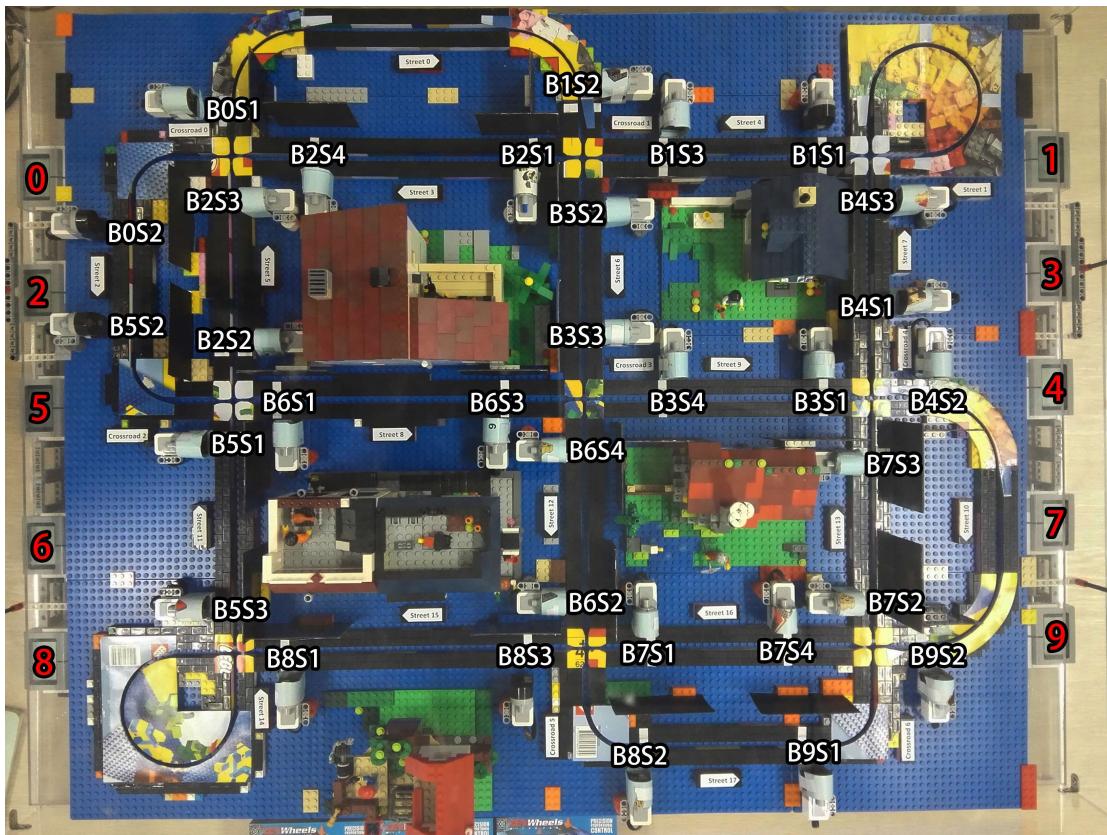


图 3-5: 红外传感器位置部署

3.3.2 位置部署

在 SimCity 实验平台搭建的迷你城市中，一共使用 10 个 EV3 程序块，编号为 0 到 9，搭载的 ev3dev 系统的 hostname 也与对应编号保持一致；共计使用 32 个 EV3 红外传感器，编号格式为 **BxSy**，表示这是连接在第 **x** 号程序块 (Brick) 的第 **y** 号输入端口上的传感器 (Sensor)。红外传感器本身并无区别，将任意传感器连接到 x 号程序块的 y 号输入端口，该传感器编号就是 **BxSy**。因此传感器编号的意义在于指明连接在 x 号程序块的 y 号输入端口的传感器应当在城市中部署的位置。

图 3-5 标明了 EV3 程序块和红外传感器的部署位置。红外传感器、城市街道和建筑都拼装在蓝色的乐高底板上，而乐高底板则放置在一块用多个柱墩架高的亚克力板上，程序块与电源插线板均收纳在亚克力板下方。乐高底板以及下方的亚克力板上事先打好了很多孔，供连接程序块与传感器的数据线穿过。为了直观地展现 SimCity 平台的实验效果，迷你城市的表层只会展示蓝牙车辆、传感器、街道等必要元素，不会暴露任何杂乱的线缆。



图 3-6: Zenwheels Micro Car

3.4 蓝牙车辆设备 ZenWheels Micro Car

SimCity 采用的蓝牙车辆设备为 Plantraco 制造的 ZenWheels Micro Car^① (以下简称小车)，如图 3-6 所示。小车长宽高的尺寸为 $54mm \times 32mm \times 25mm$ ，可以通过蓝牙在 10 米有效范围内进行远程操控。车尾拥有一个迷你 USB 接口用于充电，小车从零电量充满需要约 40 分钟，满电状态下可以持续行驶约 25 分钟。小车为后轮驱动，前轮负责转向。

3.4.1 建立蓝牙连接的方案

按下小车底盘的启动按钮可开启小车设备并被其他蓝牙设备发现，在双方配对成功后就可以建立起蓝牙连接。由于 SimCity 的控制中心在 PC 上运行，而 PC 不一定包含蓝牙模块，因此要想连接蓝牙小车就需要借助额外设备。在项目早期，我们使用了一个 Android 手机设备作为遥控器，代替 PC 连接蓝牙小车。PC 需要遥控小车时，首先通过无线网络向手机发送小车指令，手机再通过蓝牙将指令下达给小车。但这个方案有一些不足之处：(1) 利用 Android 手机作为跳板，发送的指令除了遭受蓝牙网络延迟，还要增加无线网络的延迟；(2) 需要额外启动和维护 Android 手机端应用；(3) 部分手机的低端蓝牙模块无法同时与多辆小车建立蓝牙连接。

鉴于以上缺陷，我们放弃了利用中间设备连接小车的方案，在 PC 上安装了蓝牙适配器使之具有蓝牙功能，让控制中心直接与车辆建立起蓝牙连接。我们采用的蓝牙适配器为 ORICO BTA-408^②，搭载博通 BCM20702 蓝牙芯片，传输速率达到 3Mbps，能同时与多辆小车保持蓝牙连接，足以满足 SimCity 的需求。控制中心与小车建立连接并进行断线重连的流程，在第四章会详细说明。

^①ZenWheels Micro Car. <http://zenwheels.com/>

^②ORICO BTA-408 蓝牙适配器. <http://www.orico.com.cn/goods.php?id=5218>

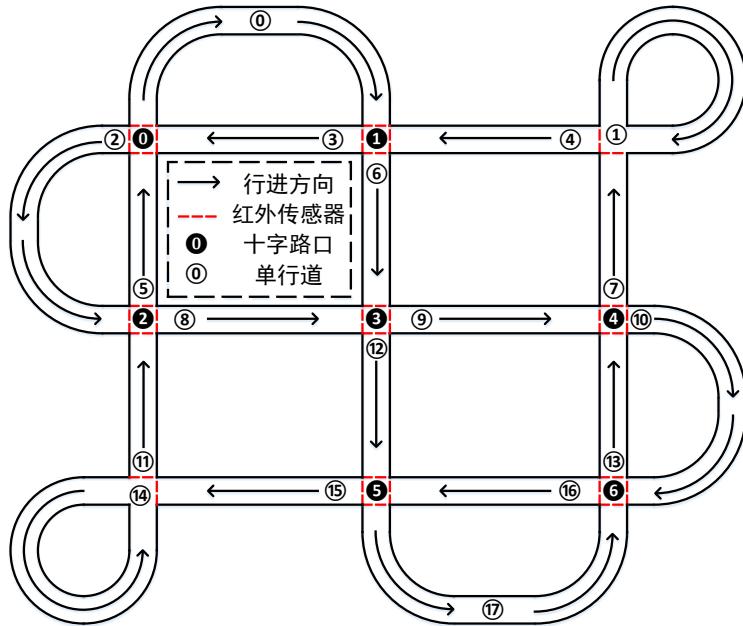


图 3-7: 城市道路设计图

3.4.2 城市街道设置

我们设计并搭建的迷你城市街道是单行道，且不允许车辆超车。道路蜿蜒曲折，纵横交错，最终形成一个首尾相接的封闭式道路，俯瞰设计图如图 3-7 所示，对应实景图为图 3-5。这样设计的目的有(1) 车辆不会驶出城市范围，一直前进也只是在反复兜圈；(2) 密集的十字路口提高了多辆车同时行驶时发生碰撞的概率，一旦一致性错误检测和修复方法处理上下文失败，不准确的上下文很可能会导致控制中心对车辆自适应调度失败而发生碰撞，从而间接提高了 SimCity 实验平台的检验效率。在每个十字路口的出入口都设有红外传感器。以传感器为界限，将整个城市街道分割成 25 个路段，包括 7 个十字路口 (Crossroad) 与 18 个单行道 (Street)，编号分别为 **Crossroad 0~6** 与 **Street 0~17**。当某个红外传感器检测到车辆经过，表示该车刚刚离开了某个路段并进入了下一个。为防止车辆发生撞车事故，我们将每个路段都视为临界区，任意时刻最多只能容纳一辆车。控制中心对车辆进行自适应调度的具体过程将在第四章作详细讲解。

为防止车辆驶出道路，我们对路面与小车也做了一些改良。观察实景图图 3-5 上的道路，可以发现我们在搭建迷你城市道路时在路面中央预留了凹槽作为导轨。相应的，我们在小车底盘上安装了金属导杆 (如图 3-8a 所示)。将导

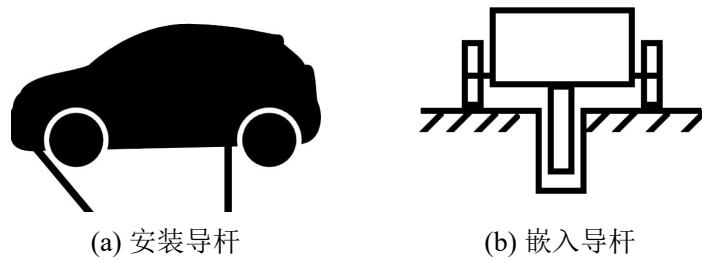


图 3-8: 改造车辆

杆嵌入路面凹槽后(如图 3-8b 所示), 车辆就会沿着导轨行驶而不会冲出道路。此外, 采用导轨限制车辆的行进还使得车辆无需偏转前轮就能顺利过弯, 简化了控制中心对车辆的操控。对于图 3-8a 中安装导杆的方式这里需要做进一步说明。我们沿着小车底盘的中轴线, 在前轮与后轮位置各安装一根导杆, 保证车头始终是顺着导轨的走向。考虑到只有后轮是主动轮, 为避免行驶过程中出现前部导杆抵住导轨, 使车尾抬起、后轮悬空, 最终车辆失去动力来源而卡住的情况, 我们对前部导杆进行了一定程度的倾斜。此时如果前部导杆抵住导轨, 反而会让车头翘起, 使车尾的后轮能紧贴地面提供动力, 从而避免了车辆卡顿。

第四章 SimCity 系统实现

本章会详细讲解 SimCity 实验平台的软件系统，包括系统中模块之间交互的层次结构，以及各个模块的功能与工作流程。

4.1 模块层次结构

SimCity 实验平台的软件系统拥有采样模块、设备连接与控制模块、一致性检测与修复模块、事件监听模块、应用模块、自适应调度模块以及状态转移模块。这些功能模块相互协作，共同支持着实验平台的正常工作。除了采样模块位于 EV3 程序块上，其余 6 个模块均运行在 PC 端，共同构成了 SimCity 的控制中心。多数功能模块可以根据设备或实验的具体要求进行定制。其中，设备连接与控制模块与一致性检测与修复模块可以通过编辑配置文件的方式直接更换设备信息以及修改一致性检测所用的一致性约束；一致性检测与修复模块、事件监听模块与应用模块均留有接口，可以替换待验证的一致性检测与修复方法，增加需要监听的事件以及添加上层的上下文感知应用。这些模块的可定制性造就了 SimCity 实验平台的可扩展化。下面将按照图 4-1 对各模块的功能以及整体的层次结构做简要介绍。

1. **采样模块：**运行在 EV3 程序块上，不间断地利用红外传感器采集新的上下文，并通过无线网络将上下文发送到 PC 端。
2. **设备连接与控制模块：**负责维护与蓝牙车辆设备以及 EV3 程序块的连接，在检测到设备断开时需要尝试重新连接，同时通知状态转移模块暂停系统的运行。该模块还负责与设备进行数据通信，包括向蓝牙车辆下达指令，以及接收程序块发来的上下文并转交给一致性检测与修复模块。
3. **一致性检测与修复模块：**根据配置文件中编写的一致性约束，对可能包含一致性错误的上下文进行约束检测，然后修复检测到的一致性错误，将一致的干净上下文上交给事件监听模块。
4. **事件监听模块：**结合最新的上下文与已有的上下文，判断系统中事件是

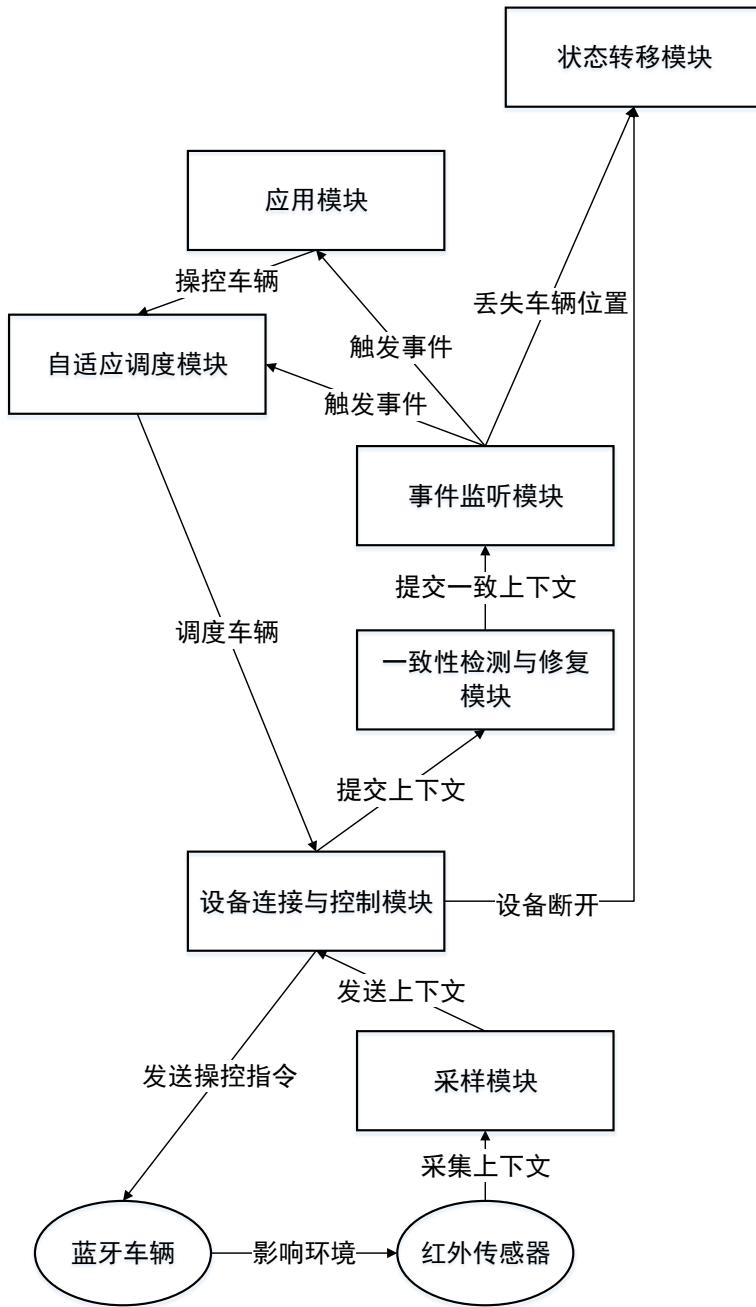


图 4-1: 模块层次结构

否触发，如某车辆驶入某路段或某设备失去连接。如果事件触发，该模块会通知监听该事件的应用模块或者自适应调度模块以作出反应。对于特殊的车辆丢失事件，该模块需要通知状态转移模块将系统切换至重定位状态。

5. 应用模块：该模块包含所有运行在实验平台上的上下文感知应用，目前

SimCity 自带一个打车应用。这些应用监听实验环境中的特定事件，并做出针对性反馈动作，例如操控车辆。

6. **自适应调度模块：**通过监听模拟城市中车辆进入路段的事件对交通状况进行分析，从而调度车辆避免相撞。
7. **状态转移模块：**拥有系统的最高权限，负责整个实验平台运行状态的转移。当收到用户或其他模块的通知时，该模块会暂停其他模块的运行，将运行状态切换至重置、暂停或重定位状态。在重置完成、设备恢复连接或车辆定位成功后，状态转移模块又会切换回上一个状态中恢复实验平台的运行。

4.2 采样模块

在 SimCity 软件系统的所有模块中，只有采样模块运行在 EV3 程序块的 ev3dev 系统上，负责 ev3dev 系统时钟的同步与连接在程序块上的红外传感器的采样工作。采样模块通过执行采样脚本 start.sh 来间接启动采样程序 sample.py。采样脚本的作用是保证最多同时只运行一个采样程序，它会首先检查是否已有采样程序在运行中，如果没有再运行新的程序。采样程序具有两个功能，一是同步 ev3dev 系统的本地时钟，二是利用红外传感器采集传感数据并发送给 PC。采样程序同步本地时钟的原因是要统一系统中每个传感数据的采样时间，以满足上下文一致性检测与修复模块中与时间相关的约束的需求。更进一步来说，某些约束不仅仅对上下文的先后顺序有要求，更是对上下文之间具体的时间间隔有限制，例如：人们进入 1 号房间的时间间隔不能低于 1 秒。这就要求本地时钟还要与真实时间进行同步，因此不能采用逻辑时钟同步算法。这里，我们采用了适用于小型局域网的物理时钟同步算法 Cristian 算法 [30]。下面对该算法的同步过程做简要介绍。

Cristian 算法需要一台时间服务器 S 和一个请求时钟同步的进程 P，同步的简要步骤如下：

1. P 向 S 发出请求，同时记录发送时间 T_{sent} 。
2. S 接收到请求后，将自己的时间 T_{server} 回复给 P。
3. P 接收到回复后，记录接收时间 T_{recv} ，计算出往返时间 $RTT = T_{recv} - T_{sent}$ ，并设置自身时钟为 $T_{server} + \frac{RTT}{2}$

Cristian 算法假设请求与回复在网络上的耗时是相等的。在低延迟的局域网中，

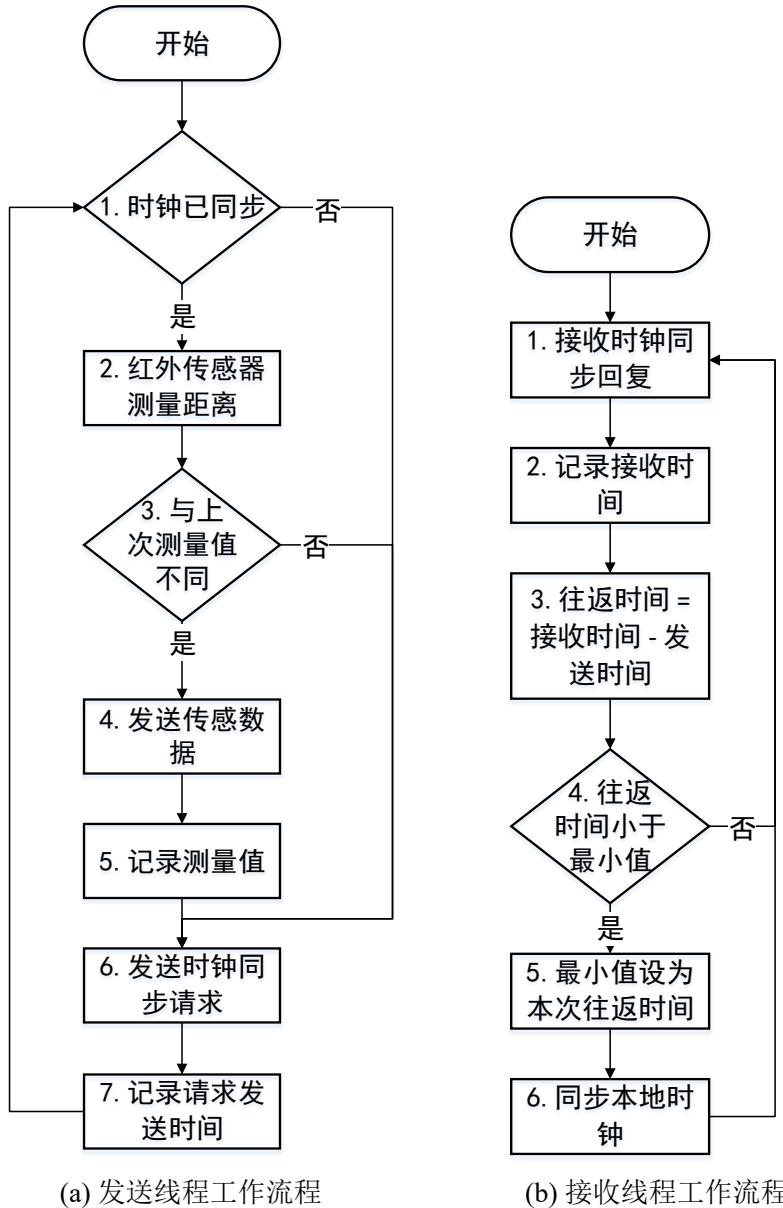


图 4-2: 采样程序工作流程图

这是一个合理的假设。为进一步提高时钟同步的准确度，进程 P 可以向 S 多次发送请求，并使用 RTT 最短的一次回复同步自己的时钟。

采样程序通过一个发送线程与一个接收线程与 PC 端的设备连接与控制模块进行数据通信。发送线程负责采集、发送传感数据以及发送时钟同步的请求，它的工作流程如图 4-2a 所示：

1. 检查本地时钟是否已同步，若还未与 PC 端同步，转 6开始时钟同步，否则转 2进行采样。

2. 利用红外传感器进行测距。转 3。
3. 判断测量值是否发生了变化。如果测量值改变，转 4 发送变化的数据，否则直接丢掉该数据，转 6。
4. 向 PC 发送关键的传感数据。转 5。
5. 记录本次测量值，以备与下个测量值比较。转 6。
6. 执行 Cristian 算法的第一步，向 PC 发送时钟同步请求。转 7。
7. 记录下同步请求的发送时间，以备接收线程计算往返时间。转 1 开始新一轮工作循环。

在步骤 3 中我们需要判断测量值的变化，再决定是否将该数据发送给 PC，原因是：(1) 发送线程中每轮循环之间的间隔设置得很小，这样能迅速捕捉到环境的变化，保证采样的及时性，但另一方面当环境未改变时也会产生大量重复的测量值，PC 对这些重复数据不感兴趣，接收到后会直接丢弃。(2) 如果 32 个红外传感器同时不间断地发送传感数据，会提高无线网络的传输延迟，而自适应调度模块通过分析过时的交通情况就可能会导致车辆调度失败。

PC 端的设备连接与控制模块在收到时钟同步请求后会执行 Cristian 算法的第二个步骤：将接收到请求时的时间戳回复给采样程序。而采样程序的接收线程负责 Cristian 算法的第三步：接收服务器的时钟同步回复，并同步本地时钟。图 4-2b 展示了接收线程的工作流程：

1. 接收来自 PC 端的时钟同步回复。转 2。
2. 记录回复的接收时间。转 3。
3. 根据发送线程记录的发送时间与接收线程记录的接收时间，可以计算出本次同步请求的往返时间 = 接收时间 - 发送时间。转 4。
4. 判断本次往返时间是否小于当前的最小值。如果比最小值小，转 5，否则放弃本次同步，转 1。
5. 更新当前最小值为本次的往返时间。转 6。
6. 同步本地时钟：本地时间 = 回复中的 PC 时间 + $\frac{\text{往返时间}}{2}$ 。转 1 开始下一轮循环。

为提高 Cristian 算法同步时钟的准确度，发送线程会不断向 PC 请求时钟同步，因此接收线程也需要不停接收对应的回复，计算往返时间，并取往返时间最短的一次回复，利用其中的 PC 时间戳与往返时间来同步 ev3dev 的本地时钟。

4.3 设备连接与控制模块

设备连接与控制模块负责 SimCity 实验平台使用的 EV3 程序块与蓝牙车辆设备同 PC 之间的连接维护与数据通信。为支撑 SimCity 正常工作，该模块需要同时与 10 个程序块以及多辆蓝牙小车保持连接。下面将详细介绍该模块的工作流程。

4.3.1 EV3 程序块的连通性检测

在 §3.2.2 节我们提到，EV3 程序块上运行着 Linux 系统 ev3dev，它通过 SSH 与 PC，更准确地说是与设备连接与控制模块进行通信。该模块使用了 JSch (Java Secure Channel)^①，一个 SSH2 安全协议的 Java 实现版本，通过执行 SSH 远程命令来维护与 ev3dev 的连接。图 4-3 展示了检测程序块连通性的流程：

1. 利用在 §3.2.2 节配置好的的 SSH 公钥与私钥，PC 尝试与 EV3 程序块建立 SSH 会话。转 2。
2. 判断 SSH 会话是否成功建立。如果会话连接建立失败，转 1 重试，否则转 3。
3. 判断 ev3dev 系统是否检测到所有已连接的 EV3 红外传感器。由于 ev3dev 会在目录 /sys/class/lego-sensor 下为检测到的传感器生成对应文件，我们利用 JSch 执行 SSH 远程命令：

```
ls /sys/class/lego-sensor
```

通过分析该目录下的文件，可以判断出是否所有传感器都准备就绪。如果有传感器未被检测到，转 4，否则转 5。

4. 利用 JSch 执行 SSH 远程命令 **reboot** 重新启动 ev3dev 系统。转 1 开始尝试重新连接。
5. 所有传感器准备就绪后，通过 JSch 执行 SSH 远程命令 **./start.sh**，尝试通过执行采样脚本来运行采样程序。转 6。
6. 利用 JSch 执行 SSH 远程命令：

```
ps -ef | grep 'python3 sample.py' | grep -v grep
```

^①JSch. <http://www.jcraft.com/jsch/>

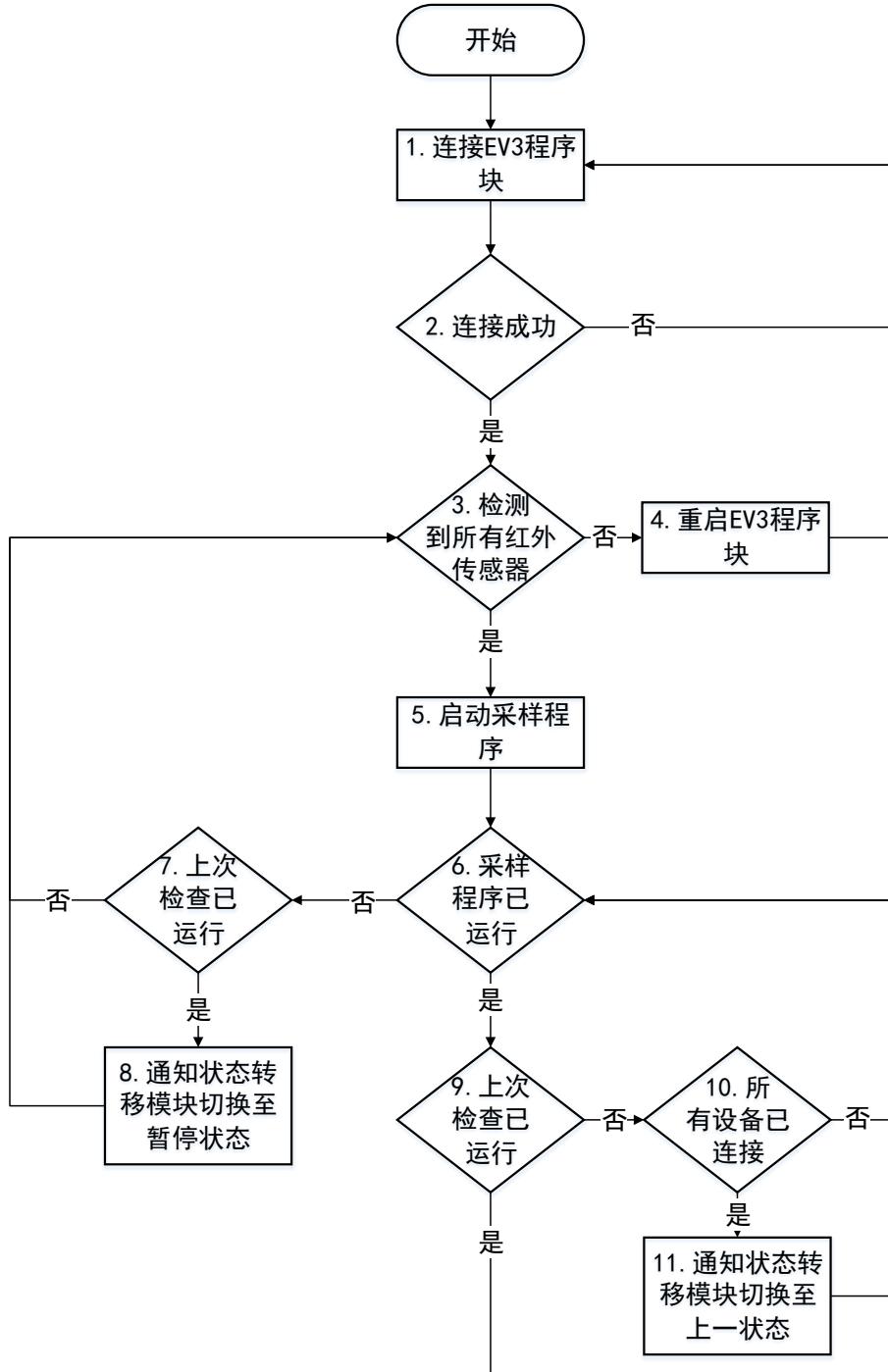


图 4-3: EV3 程序块连通性检测流程图

查询在 ev3dev 系统上采样程序 sample.py 是否已运行。如果程序未运行，转 7，否则转 9。

7. 如果上一轮检测就已经发现采样程序未运行，直接转 3从传感器连接问题

开始排除；否则表明这是首次发现采样程序未运行，设备连接状态刚刚发生了变化，需要转 8 通知状态转移模块。

8. 刚刚检测到采样程序停止了运行，程序块已经无法支持 SimCity 正常工作，因此必须通知状态转移模块将实验平台切换至暂停状态等待所有设备准备就绪。转 3。
9. 如果上一轮检测就发现程序在正常运行，表明程序块一直在正常工作，转 6 开始下一轮检测；否则表明采样程序刚刚恢复了运行，转 10 进行下一步判断。
10. 一个 EV3 程序块刚刚恢复正常工作，此时如果所有设备都处于正常运行中，就可以转 11 通知状态转移模块恢复 SimCity 到上一个运行状态；否则转 6 开始下一轮检测。
11. 刚刚检测到所有设备都恢复了正常运行，SimCity 可以再次正常工作，通知状态转移模块恢复上一个运行状态。

4.3.2 蓝牙车辆的连通性检测

设备连接与控制模块使用 BlueCove^①，一个 Java 蓝牙库与蓝牙车辆 Zen-Wheels Micro Car 建立蓝牙连接与传输指令。BlueCove 是 Java 关于蓝牙开发的规范要求 JSR-82 的一个实现版本，目前可配合 Mac OS X、WIDCOMM、BlueSoleil 以及 Microsoft Bluetooth stack 这些蓝牙驱动使用，能够满足 SimCity 利用 PC 与蓝牙小车进行通信的需求。图 4-4 展示了检测蓝牙车辆连通性的流程：

1. 根据蓝牙车辆设备的蓝牙串口传输服务 URL，尝试与车辆建立蓝牙连接。转 2。
2. 如果与蓝牙车辆成功建立连接，该设备会定时发送随机数据作为心跳包。因此我们对连接使用阻塞读来判断连接是否断开。如果阻塞读抛出异常，表明连接失败，转 3；如果读到心跳，表明连接成功，转 5。
3. 本轮检测结果是连接断开，如果上一轮检测结果也一样，表示当前车辆状态未变化，直接转 1 尝试重连；否则表示车辆连接断开是刚刚发生的事情，需要转 4 通知状态转移模块进行处理。
4. 由于检测到新的设备状态变化——设备断开，此时 SimCity 已不具备正常工作的条件，因此需要通知状态转移模块切换实验平台至暂停状态，等

^①BlueCove. <http://bluecove.org/>

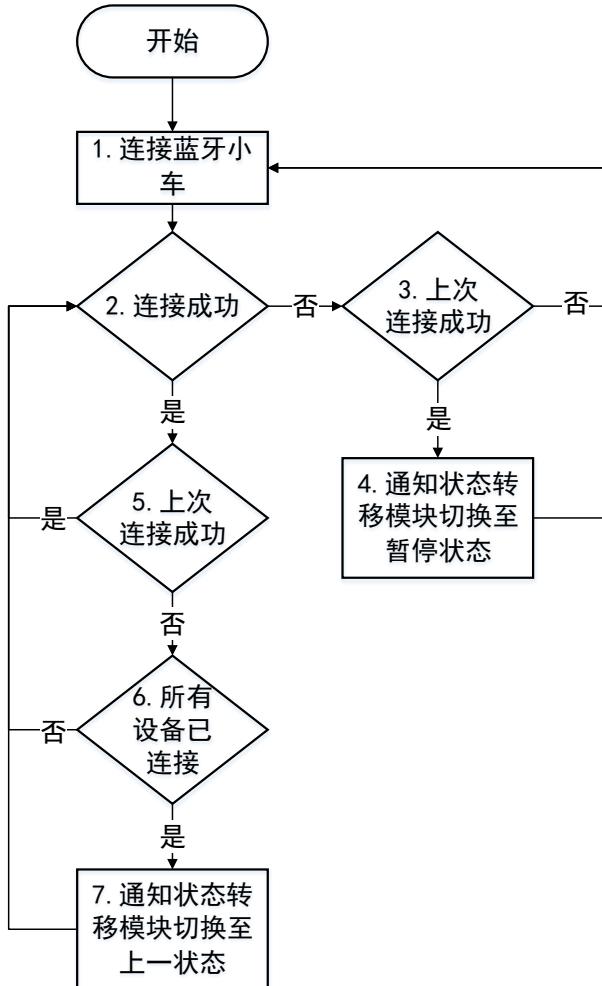


图 4-4: 蓝牙车辆连通性检测流程图

待设备恢复连接。转 1。

5. 本轮检测结果是连接正常，如果上一轮的检测结果相同，表示设备状态未改变，直接转 2 开始下一轮检测；否则表示车辆刚刚恢复连接，需要转 6 进行进一步判断。
6. 一个车辆设备刚刚重连成功，此时如果所有设备都连接正常，那么 SimCity 又能够正常工作了，转 7 通知状态转移模块；否则转 2 继续检测。
7. 首次检测到所有设备都已正常连接，必须通知状态转移模块恢复实验平台到上一状态，继续正常运行。然后转 2 进行下一轮检测。

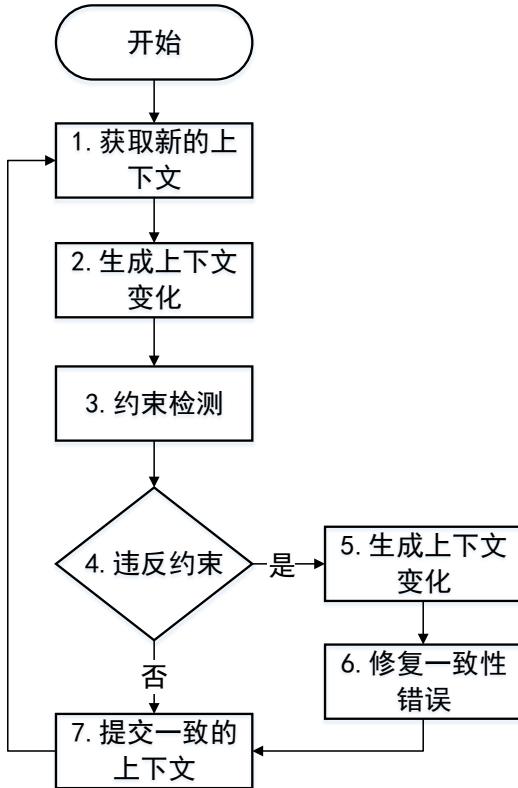


图 4-5: 一致性检测与修复流程图

4.4 一致性检测与修复模块

一致性检测与修复模块通过读取配置文件，建立一致性约束。对于下层提交的上下文，该模块首先使用指定的约束检测方法检测是否有约束被违反，一组上下文中是否包含一致性错误。如果上下文不一致，需要按照指定的修复策略对上下文进行修复。最后将一致的上下文交付上层使用。

由于上下文一致性错误处理的方法在 §2.1.1 节已经详细介绍过，因此本节仅根据图 4-5 分步讲解一致性检测与修复模块的工作流程：

1. 模块获取到下层传来的一个新上下文。转 2。
2. 对于该上下文符合的所有模板（上下文集合），生成一组上下文变化（Context Change）。在 §2.1.1 节中我们提到上下文变化是三元组 (*type*, *pattern*, *context*)，这里的一组上下文变化中，*type* 取值 *addition*，*pattern* 为各个符合要求的集合，*context* 为该上下文实例。对于某些达到最大容量的集合，还需要额外生成一个删除最早上下文的上下文变化。转 3。
3. 根据上下文变化，对受影响的约束对应的一致性计算树增加新分支或剪

- 枝，接着计算真值，最后生成一致性链。转 4。
4. 如果计算出的真值为 *True*，表示一致性约束未被违反，参与检测的上下文是一致的，可以转 7 进行提交。否则需要转 5 修复一致性错误。
 5. 根据修复策略决定需要删除或者添加的上下文，再生成一组上下文变化。例如，如果修复策略为 Drop-latest，那么需要生成一组删除最新上下文的上下文变化，与之前 2 中生成的一组上下文变化的 *type* 正好相反。转 6。
 6. 仿照 3，根据上下文变化更新一致性计算树的结构以及各个节点的真值与一致性链，使得对应约束的真值恢复 *True*。转 7。
 7. 向上层提交不含一致性错误的上下文，然后转 1 开始下一轮检测。

可以发现，一致性检测与修复模块在 SimCity 系统中是一个中间件，利用约束检测与一致性错误修复过滤掉原始上下文中的一致性错误，尽可能地排除噪声或添补可能缺失的信息，将准确一致的上下文交付上层使用，避免应用运行出错。

4.5 事件监听模块

事件监听模块负责监听 SimCity 系统内部与外部的事件，并在事件触发时通知感兴趣的其他模块。例如，根据收到的上下文信息，该模块会触发车辆进入路段事件，并通知自适应调度模块及时对车辆进行安全调度。事件监听模块的工作流程参考了观察者设计模式 (Observer Pattern)^①，我们结合 UML 类图 (图 4-6) 来进行解释。

EventManager 管理观察者模式中的主题 (Subject)——事件 (Event)，并保存对不同主题感兴趣的观察者对象 (Observer)。接口 EventListener 代表了抽象观察者，当需要观察某个主题时 (对某个事件感兴趣)，一个具体观察者需要 (1) 实现抽象观察者 EventListener 接口，即方法 eventTriggered()，告知 EventManager 事件触发时应当对自己采取的操作；(2) 调用 EventManager.register()，注册自己感兴趣的一个或多个事件。当系统发生某事件时，会通过调用 EventManager.trigger() 唤醒工作线程 EventManager.worker。工作线程随后从观察者集合 EventManager.listeners 中取出所有对刚刚产生的事件感兴趣的观察者 EventListener，并依次调用 EventListener.eventTriggered()，及时给这些观察者发

^① 观察者模式. https://en.wikipedia.org/wiki/Observer_pattern

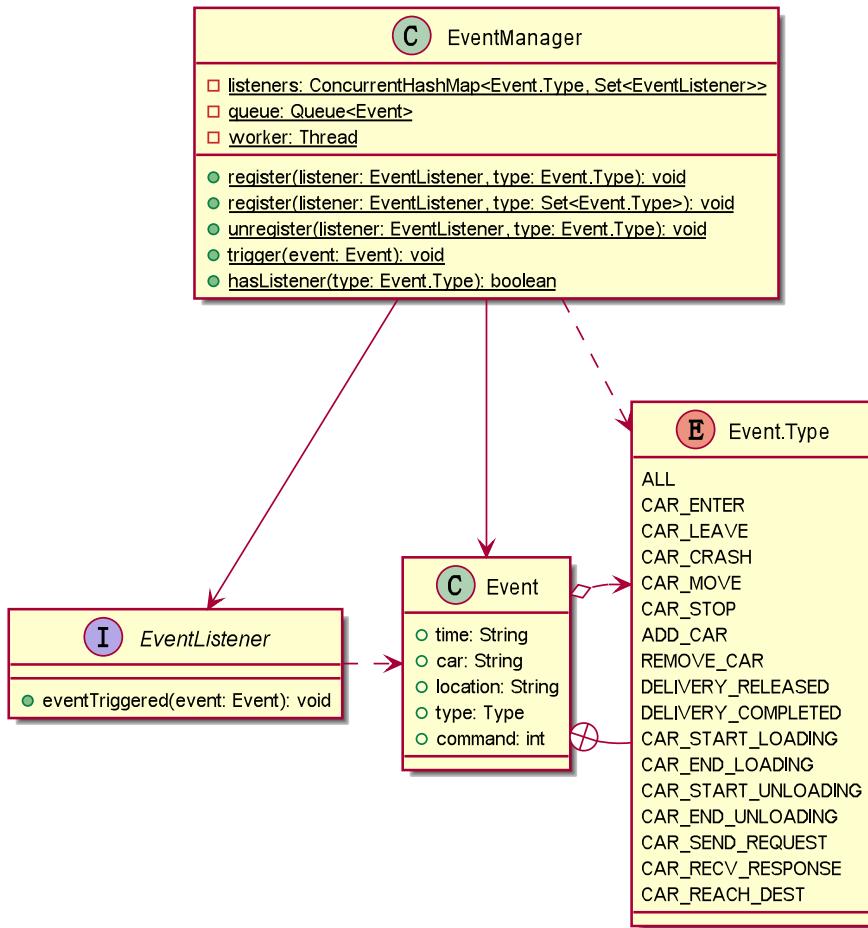


图 4-6: 事件监听模块 UML 类图

出通知。

4.6 应用模块

应用模块用以管理所有运行在 SimCity 实验平台上的上下文感知应用。上下文感知应用可以在事件监听模块注册感兴趣的事件，并在事件触发时做出一些反应，例如向自适应调度模块发送操控车辆的请求或者在用户界面显示特定信息等。应用模块是上下文的最终使用者，下层模块会源源不断地向上层应用提供新的上下文，而应用会根据这些上下文做出不同反应。当一致性检测与修复模块出错时，含有不一致错误的上下文就会被提交给应用，而应用很可能据此做出错误决定，最终将错误反馈到车辆行动或是用户界面上。这样，我们就能够发现一致性检测和修复方法的问题。

应用模块中自带一个打车应用，可以自动生成打车任务，也可以通过用户界面指定接客点与目的地，发布自定义任务。打车应用会寻找离接客地点最近的空闲车辆，将任务分配给它。车辆完成打车任务的过程如下：

1. 空闲车辆接到打车任务，遵守自适应调度模块的调度向前安全行驶。
2. 当车辆行驶到接客地点时，打车应用向自适应调度模块下达停止该车的命令。
3. 当车辆在接客地点停留一段时间后，打车应用视为顾客已经上车，向自适应调度模块下达启动车辆的命令。
4. 当车辆行驶到目的地，打车应用再度向自适应调度模块下达停止该车的命令。
5. 当车辆在目的地停留一段时间后，打车应用视为顾客已下车，将该打车任务标为已完成状态，同时将车辆切换至空闲状态，等待接受下一个打车任务。

打车应用通过自动发布打车任务，不断向自适应调度模块发送操控各个车辆的请求，令车辆不停在前进与停止之间切换，表现出物理环境变化的复杂性，确保对一致性检测与修复方法进行充分的考验。只要有一个一致性错误未被检测到或是未被正确修复，它就极有可能造成应用的错误反应或是自适应调度模块的错误调度，最终导致应用任务无法完成甚至撞车事故。

打车应用会在第五章中进行演示与说明，这里不做展示。

4.7 自适应调度模块

自适应调度模块对车辆进行自适应调度的目的是防止车辆相撞，保障它们安全行驶，顺利完成任务。自适应调度的思路是将传感器之间的路段视为一个临界区，车辆进入路段之前需要先申请该路段的唯一令牌 (token)，得到令牌后方可进入。而如果没有获得令牌，就需要停下车辆进行等待。当车辆离开路段时需要归还令牌，此时如果该路段还有其他车辆等待进入，就把令牌发给等待最久的车辆，允许其进入。这里需要解释一下，由于路段是由多个传感器为界限划分形成的区域 (§3.4.2 节)，因此车辆经过某个传感器进入一个路段，也可以看作是经过该传感器离开另一个路段，这两个事件是同时发生的。

自适应调度模块一方面向事件监听模块注册监听车辆进入路段事件和车辆离开路段事件，实时更新交通状况，另一方面根据最新的交通情况处理来自上

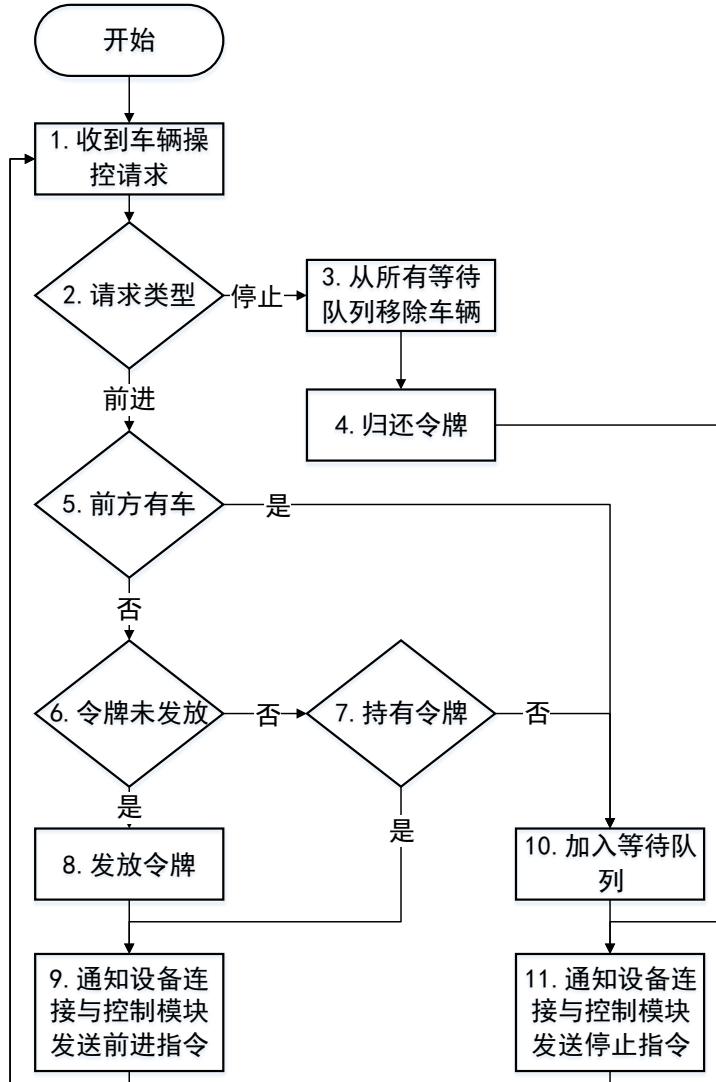


图 4-7: 自适应调度流程图

层应用模块或用户界面的操控车辆请求。得益于§3.4.2节中的城市街道导轨与车辆导杆的设计，我们只需要前进和停止两种指令就可以操控车辆经过所有的路段和红外传感器。因此自适应调度模块也只需对这两种操控请求进行处理。图4-7展示了该模块的处理流程：

1. 自适应调度模块收到一个来自上层应用或者用户界面的车辆操控请求。
转1。
2. 判断请求类型，如果是车辆停止请求，转3，如果是车辆前进请求，
转5。
3. 对于停止请求，车辆将不会再进入其他路段，因此从所有路段的等待队

- 列中移除该车辆。转 4。
4. 因为车辆不会再进入其他路段，所以需要归还它持有的所有路段的令牌，供其他等待车辆使用。转 11。
 5. 对于前进请求，首先判断前方路段是否有车。如果有车，转 10 进行等待，否则转 6 进行下一步判断。
 6. 如果该空闲路段的令牌还未发放，转 8 发放令牌，否则转 7 进行下一步判断。
 7. 如果请求的车辆正好持有该路段的令牌，那么跳过 8 直接转 9 发送指令。否则表明有其他车辆先一步申请到了该路段的令牌，只有转 10 进行等待。
 8. 将空闲路段的令牌发给请求前进的车辆。转 9。
 9. 自适应调度模块向下层的设备连接与控制模块下达车辆前进指令。车辆接收到含有前进指令字的指令后将会向前行驶。转 1 进入新一轮工作循环。
 10. 请求前进的车辆不满足进入路段的条件，因此先加入该路段的等待队列，再转 11 发送停止指令。
 11. 向设备连接与控制模块发送停止指令，命令车辆立即停止行驶。转 1 开始处理下一个车辆操控请求。

自适应调度模块还需要维护路段的等待队列。当车辆离开路段事件触发时，自适应调度模块会从该空闲路段的等待队列中取出最早等待的车辆，将令牌发放给它，并通知设备连接与控制模块发送该车的前进指令。

4.8 状态转移模块

状态转移模块是 SimCity 软件系统中权限最高的模块，负责在系统内部或外部的特定事件触发时，切换实验平台的运行状态，改变各个模块的运行模式。该模块会在初始化、正常运行、重置、重定位以及暂停这五个状态之间进行切换，如图 4-8 所示。下面对每个运行状态的转移条件以及系统在该状态下的行为进行讲解。

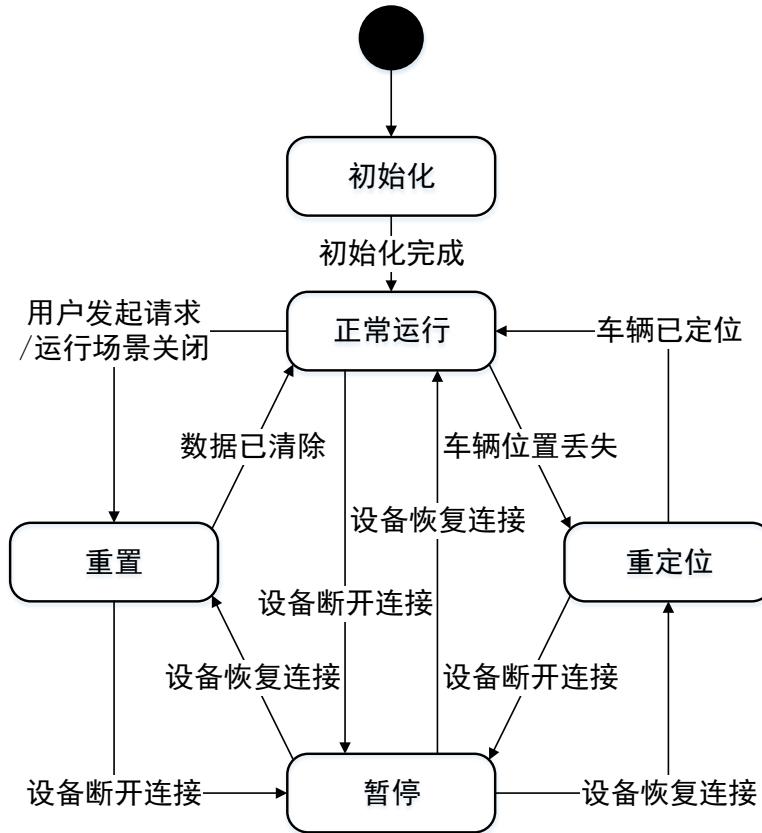


图 4-8: 状态转移模块 UML 状态图

4.8.1 初始化状态与正常运行状态

启动 SimCity 系统后，它首先进入初始化状态。设备连接与控制模块会读取 EV3 程序块、蓝牙车辆的设备配置文件，尝试与设备建立连接，并远程启动程序块上的采样程序，让采样模块也开始工作。在初始化状态下，设备连接与控制模块只会响应采样程序的时钟同步请求，对于收到的上下文信息会直接丢弃而不会交给上层模块。一致性检测与修复模块会读取模板 (Pattern) 与一致性约束的配置文件，建立一致性约束，进入就绪状态。事件监听模块此时会登记各模块感兴趣的事件。应用模块与自适应调度模块各自启动工作线程并准备就绪。待所有模块都正常运行后，初始化完成，系统切换至正常运行状态。

在正常运行状态下，系统可以选择开启三种运行场景 (Scenario)，分别是理想场景 (Ideal)、包含错误的场景 (Noisy) 以及修复错误的场景 (Fixed)。在理想场景下，SimCity 会展现在一个上下文不存在一致性错误的理想世界中的运行状况。此时所有模块都正常工作，一致性检测与修复模块会过滤掉上下文中的一致性错误，将一致的上下文交给上层模块使用。但是上层模块不会收到一

致性错误的检测与修复信息，在用户界面中也不会显示这些信息，用户会观察到 SimCity 理想的运行状况。在包含错误的场景中，一致性检测与修复模块只会检测一致性错误而不进行修复，因此上层模块只能收到一致性错误的检测信息，使用不一致的上下文。此时 SimCity 实验平台会详细展示出不一致的上下文是怎样导致应用出错，并最终导致撞车事故的。在最后的修复错误的场景中，一致性检测与修复模块正常工作，上层模块会收到完整的一致性错误的检测与修复信息以及一致的上下文。SimCity 恢复正常运行，并在用户界面上实时显示检测与修复的一致性错误。

系统在未开启任何运行场景时处于空闲状态，此时设备连接与控制模块的工作模式与在初始化状态下相似，只回复采样模块的时钟同步请求而丢掉采集的上下文数据。由于上层模块不会收到新的上下文，因此它们也处于空闲状态。

我们将在第五章对 SimCity 实验平台的用户界面以及各项功能进行演示与说明。

4.8.2 重置状态

SimCity 系统在三种运行场景的工作过程中，会产生很多运行时数据，例如一致性检测与修复模块中的上下文集合，或者打车应用发布的打车任务等等。而重置正是为了清除各个模块中的运行时数据，恢复最初的空白状态。当系统处于正常运行状态下，在任意时刻，用户都可以通过用户界面发起重置请求，清空系统的运行时数据，来重新启动或切换运行场景。在关闭运行场景时，状态转移模块也会自动切换到重置状态，清除场景中残留的信息，为启动下一个场景做准备。当场景清理完毕后，状态转移模块会自动切回正常运行的空闲状态。重置还有一个可选的车辆定位功能。在初始化完成后，系统不知道各辆车的位置，此时需要先切换至重置状态，对车辆进行依次定位，然后切换回正常运行状态。

在 §4.1 节我们提到，SimCity 系统中的多个模块之间交互频繁，且模块内部又有工作线程在不断产生与处理运行时数据。如果在模块正常工作时直接清理数据，很可能会干扰工作线程处理数据的正常流程，或是破坏一些非线程安全的数据类型，最终导致工作线程运行异常。有鉴于此，重置过程分为 4 个步骤：(1) 将系统切换到重置状态，禁用用户界面，暂停模块的运行，停止所有车辆；(2) 清除运行时数据；(3) 定位车辆(可选)；(4) 启用用户界面，切换系统至

```

1 class Worker extends Thread {
2     private Queue<Request> queue = new LinkedList<Queue>();
3
4     public void run() {
5         StateSwitcher.register(Thread.currentThread());
6         while(true) {
7             synchronized (queue) {
8                 while(queue.isEmpty() || !StateSwitcher.isNormal()) {
9                     try {
10                         queue.wait(); //<--- safe point
11                     }
12                     catch (InterruptedException e) {
13                         if(StateSwitcher.isResetting() && !StateSwitcher.
14                             isThreadReset(thread)) {
15                             queue.clear();
16                             StateSwitcher.resetThread(thread);
17                             if (StateSwitcher.allReset())
18                                 StateSwitcher.wakeUp();
19                         }
20                     }
21                 }
22             }
23             Request request;
24             synchronized (queue) {
25                 request = queue.poll();
26             }
27
28             //process this request
29         }
30     }
31
32     public void add(Request request) {
33         if(StateSwitcher.isResetting())
34             return;
35         synchronized (queue) {
36             queue.add(request);
37             queue.notify();
38         }
39     }
40 }

```

图 4-9: 工作线程

正常运行状态并恢复模块的运行。需要指出，重置需要暂停与恢复运行的模块不包括底层的采样模块和设备连接与控制模块，因为它们还需要为重置功能提供上下文来源与控制车辆，当然也不包括提供重置功能的状态转移模块。

第一步，状态转移模块将当前的系统状态从 **NORMAL** 改为 **RESETTING**，禁用了用户界面上所有的 UI 元素，并直接命令设备连接与控制模块停下所有的车辆。暂停模块运行的难点在前面已经提到，下面将对解决方法进行详细说明。系统中的各个模块都有自己的工作线程，模块之间的交互

是通过发送请求到工作线程的队列中等待处理的方式实现的。图4-9显示了工作线程的典型框架 (Skeleton)。其他模块通过调用第32行的`worker.add()`向该工作线程的工作队列`queue`中 (第2行) 添加请求。工作线程首先向状态转移模块`StateSwitcher`注册自己 (第5行)，然后进入无限的工作循环中 (第6行 ~ 第29行)。工作循环的逻辑一般是先检查工作队列是否为空 (第8行)，如果没有待处理的请求就进入等待 (第10行)，否则就取出队列中的第一条请求 (第25行) 进行处理 (第28行)，处理完毕后回到第6行开始新一轮循环。

当工作线程执行到第10行时会停止工作，陷入等待，我们称之为安全点 (Safe Point)，因为此时可以安全地清理工作队列`queue`中的运行时数据。为了让所有的工作线程达到安全点，我们对他们依次调用`Thread.interrupt()`进行中断。该方法会设置线程的中断状态 (Interrupt Status)，并对不同运行状态中的线程产生不同效果^①，这里只讲解文中涉及到的情况。对于阻塞在`object.wait()`，`thread.join()`以及`Thread.sleep()`中的线程，它会抛出`InterruptedException`，清除中断状态。而对于未阻塞的线程，它会继续执行，直到遇到`object.wait()`，`thread.join()`以及`Thread.sleep()`时会直接抛出`InterruptedException`并清除中断状态。

再回顾前面的代码框架，工作线程在正常运行时有两种状态：已阻塞在第10行的`queue.wait()`上和未阻塞的。对于已阻塞的线程，虽然它已经处于安全点，但是既没有清空自己的工作队列，也没有通知状态转移模块它已经处于安全点，因此还需要再次唤醒它。调用该线程的`interrupt()`，它会抛出`InterruptedException`，然后在第12行被捕获。在第14行~第17行，工作线程会清空自己的工作队列，向状态转移模块报告自己已经重置。如果所有线程都已重置，它还会通知模块进入下一阶段。跳出`catch`语句块后，控制流回到第8行，由于此时队列被清空，`queue.isEmpty()`返回`true`，线程顺利地再次进入安全点。对于未阻塞的线程，它的中断状态被设置。控制流继续执行，最终到达第8行。由于模块已切换到重置状态，`!StateSwitcher.isNormal()`返回`True`，线程进入第10行的安全点。由于被设置了中断状态，线程直接抛出`InterruptedException`，被捕获进入`catch`语句块。后续的处理流程与阻塞的线程相同，不再赘述。综上，我们只需要调用工作线程的`interrupt()`进行中断，无论线程此时处于何种状态，都会安全地清空自己的工作队列，并通知模块，最后进入安全点暂停工作。

^①Java 线程中断机制. <https://docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html>

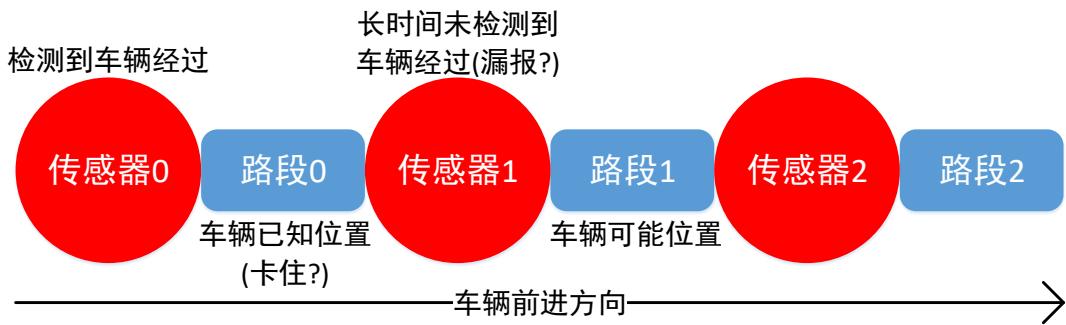


图 4-10: 重定位示意图

待所有模块的工作线程都进入安全点后，重置过程进入第二步——清除运行时数据。在上一步中，工作线程在暂停前已经清空了自己的工作队列。因此这一步会继续清除其他的运行时数据，包括：路段的令牌、等待队列，车辆的指令、状态、打车任务，一致性检测与修复模块的上下文集合、一致性计算树，打车应用的选项、任务，用户界面所有 UI 元素保存的选项与记录等等。

第三步的定位车辆是一个可选步骤，如果所有车辆的位置已知，这一步就不必执行。重置的定位功能会利用所有的红外传感器对车辆进行定位。在第一步中状态转移模块已经停下了所有车辆，因此所有传感器不会再检测到有车辆经过。接着模块只启动一辆车，当有传感器第一个报告有车辆经过时，那必定是唯一启动的这辆车，最终将该车定位到该传感器沿着前进方向的下一路段上，并停止该车。如果还有其他位置未知的车辆，模块重复上述过程一一进行定位。

最后一步，状态转移模块重新启用用户界面，将系统状态从 **RESETTING** 切换至 **NORMAL**，并不开启任何运行场景，处于空闲状态中。此时虽然所有模块的工作线程依旧在图 4-9 中第 10 行的安全点等待，但是当其他模块通过 `Worker.add()` 向工作队列 `queue` 添加请求并唤醒工作线程后（第 32 行），因不满足第 8 行的条件，线程会跳出等待循环，开始正常工作。

4.8.3 重定位状态

在系统正常运行，车辆川流不息的过程中，可能会出现这样一种状况：以图 4-10 为例，一辆车经过传感器 0 后进入路段 0。车辆继续向前行驶，但迟迟未被后面的传感器 1 检测。产生这种现象有两种可能，一是车辆由于电量损耗动力减小或者导杆抵住导轨而卡在路段 0 中，前进受阻；二是车辆已经驶过了传感器 1 进入了路段 1，但是传感器 1 发生了漏报错误 (False Negative)。不幸

的是，由于 SimCity 系统只依靠传感器来感知车辆，没有其他手段检测车辆的真实位置，一致性检测与修复模块无法判断是否真的有上下文缺失，因而不能进行正确的修复。为了处理这类特殊的车辆位置丢失现象，状态转移模块为每辆车在每个路段都设置一个通行时间上限 `timeout`，当车辆在路段中处于前进状态时就会进行计时，而进入新的路段时又会复位并重新计时。当车辆在路段中行驶的累计时间超过上限，我们视为系统丢失了该车的位置，状态转移模块随后中断系统的正常运行，切换到重定位状态进行处理。

重定位的过程分为三个步骤：(1) 切换系统至重定位状态，中断模块的运行，停止所有车辆，禁用用户界面；(2) 重定位丢失的车辆；(3) 恢复用户界面，恢复车辆运行，切换系统至正常运行状态，恢复模块的运行。和重置状态一样，这里暂停和恢复运行的模块也不包括采样模块和设备连接与控制模块。

第一步，状态转移模块首先将系统状态从 **NORMAL** 切换到 **RELOCATING**。再观察图 4-9 的工作线程的框架，已阻塞在 `queue.wait()` 安全点（第 10 行）的线程不受影响，未阻塞的线程执行到第 8 行会由于满足 `!StateSwitcher.isNormal()` 条件而携带未完成的请求强制进入安全点。因为我们只是暂停线程的运行，因此不能调用 `Thread.interrupt()` 清除工作队列中的请求。接着状态转移模块直接要求设备连接与控制模块停止所有车辆，然后禁用用户界面的所有 UI 组件。

第二步重定位车辆是最关键的步骤。和重置的定位功能类似，我们仍然是采用传感器检测唯一启动的车辆的方式进行定位。但与它不同的是，我们已经获得了丢失位置的车辆的大致位置（图 4-10 中的路段 0 或路段 1），因此只需监听附近的而非所有的传感器就能进行定位。再回顾图 4-10，车辆在路段 0 中，根据自适应调度模块的临界区原则，路段 0 中肯定没有其他车辆。而车辆又是向路段 1 前进的，根据图 4-7，该车必然已经获得了路段 1 的令牌，并且路段 1 中也没有车。综上，路段 0 和路段 1 中只有待重定位的车辆，它在这两段路中行驶不会碰撞到其他车辆。根据这一推断，我们提出了后向重定位算法，以图 4-7 为例进行解释：

1. 设备连接与控制模块向待重定位的车辆发送后退命令，让它逆向行驶，并开始计时。
2. 因为车辆只可能位于路段 0 或路段 1 上，所以开始监听传感器 0 与传感器 1。
3. 在设定的时限内，如果传感器 1 报告有车辆经过，表明该车位于路段 1，

因此将其重定位到路段 1 上，并立即停止车辆；如果传感器 0 检测到车辆经过，同理将该车定位到路段 0 上并停止车辆。

4. 如果到达时限后两传感器均未检测到车辆，则自动重定位失败。这同样有多种可能的原因，一是车辆卡在路段 0 或者路段 1 上，倒退受阻；二是传感器 0 或传感器 1 再次发生漏报。在这种极端情况下，用户界面上会弹出提示，请用户手动将车辆放置在路段 0 上并排除阻滞现象。

对比城市道路实景图图 3-5 和设计图图 3-7，可以发现有不少路段是弯道。虽然图 3-8 中对车辆和路面的改装可以让车辆在前进时顺利过弯，但是车辆如果是逆向行驶，基本不可能通过弯道路段。此时后向重定位就极有可能失败，只能依赖用户进行手动重定位。为了提高自动重定位的成功率，提升 SimCity 的运行效率，当路段 0 和路段 1 都是直道时，我们会调用后向重定位算法。而当路段 0 或路段 1 为弯道时，我们又提出了更复杂的前向重定位递归算法进行处理。下面再次以图 4-10 为例，并结合图 4-11 中该算法的代码框架进行解释。

对于丢失位置的车辆 (`isLost = true`)，它既可能位于路段 0 (`road0`) 也可能位于路段 1 (`road1`)，因此在重定位中向前行驶时，它可能经过传感器 1 (`sensor1`) 到达路段 1，也可能经过传感器 2 (`sensor2`) 到达路段 2 (`road2`)。由于前面的推断，我们知道实际上在 `road0` 与 `road1` 上只有待重定位的车辆，且该车在系统层面上位于 `road0`。因此第 5 行到第 6 行将 `road1` 中所有车辆前移目前不起任何作用。然后考虑到车辆可能位于 `road1`，前向重定位时会进入 `road2`，为避免车辆碰撞，需要清空 `road2` 中的车辆。在第 10 行到第 11 行，`road2` 中的车辆会被依次选中，并递归调用前向重定位算法进行前移。前方路段被清空后，方可启动车辆前进（第 13 行）。对于前移的、未丢失的车辆 (`isLost = false`)，如果采用“坐标转换”将它的确定位置同样定在图 4-10 的 `road0`，那么它只可能经过 `sensor1` 到达 `road1`，因此只需监听 `sensor1`（第 14 行）。而对于丢失的车辆，还需要监听 `sensor2`（第 16 行）。然后在第 19 行算法等待一段时间，如果任意监听的传感器检测到车辆经过，立即调用 `Thread.interrupt()` 终止等待。算法随后停下车辆（第 23 行），并根据报告的传感器进行定位：(1) 在等待时间内未有传感器检测到车辆（第 24 行），重定位失败，在第 25 行进行手动重定位；(2) 在限定时间内有传感器及时报告，直接将车辆定位到该传感器的下一路段（第 27 行）。

仔细观察图 4-11，会发现前向重定位递归算法的收敛条件是：(1) 对于未丢失的车辆 (`isLost = false`)，前方 `road1` 无车；(2) 对于丢失的车辆 (`isLost =`

```

1 void forwardRelocateRecursively(Car car, boolean isLost, Sensor sensor) {
2     Sensor sensor0 = sensor, sensor1 = sensor.nextSensor, sensor2 = sensor1
3         .nextSensor;
4     Road road0 = sensor0.nextRoad, road1 = sensor1.nextRoad, road2 =
5         sensor2.nextRoad;
6     //drive away all cars in road1, in case the lost car enters road1
7     while (road1.hasCars())
8         forwardRelocateRecursively(road1.getFirstCar(), false, sensor1);
9
10    if (isLost)
11        //drive away all cars in road2, in case the lost car enters road2
12    while (road2.hasCars())
13        forwardRelocateRecursively(road2.getFirstCar(), false, sensor2);
14
15    car.driveForward();
16    listen(sensor1);
17    if (isLost)
18        listen(sensor2);
19
20    try {
21        Thread.sleep(TIMEOUT);
22    }
23    catch (InterruptedException e) {}
24
25    car.stop();
26    if (detectedSensor == null)
27        relocateManually(); //time out, relocation failed
28    else
29        car.setLocation(detectedSensor.nextRoad);
30 }

```

图 4-11: 前向重定位递归算法

true), 前方 road2 无车 (根据推断 road1 必然无车)。如果当前车辆不满足收敛条件, 首先会递归调用该算法处理更前方路段的车辆。如此一路递归调用下去, 算法真正的处理顺序反而会变为, 先依次将最前方妨碍后续定位的车辆前移, 再重定位最后的车辆。因此, 前向重定位算法在提高重定位成功率的同时, 也保证了重定位过程中车辆的安全性。

最后是重定位的第三步。状态转移模块首先恢复了用户界面的使用, 接着恢复所有车辆原本的行驶状态, 然后切换系统至 **NORMAL**, 最后对所有进入安全点的工作线程通过调用 `Thread.interrupt()` 进行唤醒, 恢复模块的正常工作。

4.8.4 暂停状态

不管系统当前处于正常运行状态、重置状态还是重定位状态, 当设备连接与控制模块检测到有任何设备断开时, 都会立刻通知状态转移模块暂停系统运

行。在所有设备再次正常工作后，状态转移模块再恢复系统到上一个状态继续运行。在暂停状态，除了采样模块与设备连接与控制模块还在工作，其他高层模块都会暂停工作。暂停和恢复系统运行的操作也与重置以及重定位的第一步与最后一步相似。暂停系统运行首先将系统状态从 **PREV_STATE** 切换到 **SUSPENDED**，然后停下所有车辆，最后禁用用户界面。而恢复系统运行则是先重新启用用户界面，接着启动原本处于行驶中的车辆，再将系统状态切换回 **PREV_STATE**，最后对所有阻塞的工作线程调用 `Thread.interrupt()` 进行唤醒。

第五章 实验与分析

在本章，我们首先对 SimCity 实验平台的实验设置进行说明，然后介绍用户界面的功能，接着结合用户界面与实景图对三种运行场景进行分别演示，最后列出实验数据对本实验平台的性能进行分析。

5.1 实验设置

如图 3-5 所示，我们共使用了 32 个 EV3 红外传感器，分别连接在 10 个 EV3 程序块上。每个程序块上都运行一个采样程序，不间断地使用该程序块上连接的 2~4 个传感器采集上下文数据，并选择关键性数据传给 PC 端的控制中心。控制中心会同时控制 1~3 辆蓝牙车辆，在尺寸为 $100\text{cm} \times 90\text{cm}$ 的模拟城市内穿行并互相避让，安全地完成打车应用分派的打车任务。

对于上下文一致性检测与修复模块，我们选用了具有代表性的局部约束检测 (Partial Constraint Checking, Pcc) [2] 作为一致性检测方法。至于修复策略，我们选择了 Drop-latest，这是因为在重置的定位阶段，我们在确定车辆的位置后，会产生一条正确的初始上下文，对应于车辆的真实位置。也就是说，初始的上下文中不包含一致性错误。如果添加一个新的上下文后，它与旧的一致的上下文反而违背了一致性约束，那么这个新的上下文极有可能是不准确的，所以才会与本来一致的上下文检测出一致性错误。因此，我们采用 Drop-latest 策略丢弃最新的上下文，修复掉一致性错误。

在实验中，我们共使用了两类模板，如图 5-1 所示。一类模板是通用模板，数量只有一个，名为 latest。它的大小为 1，对上下文的所有域都不加限制。因此只要有新的上下文产生，都会加入该模板中。又因为它的容量为 1，所以模板中始终只存放最新的一个上下文，故名为 latest。另一类模板是车辆模板，每辆车都会有一个对应的该模板。它可以容纳 2 个上下文，只对上下文的车辆名称进行限制。因此不同车辆的上下文会被加入对应的模板中。

我们建立了三类一致性约束，如图 5-2 所示。这里，我们用一个模板的 id 来表示该模板对应的上下文集合。约束 A(图 5-2a) 是唯一一个通用约束，与

```
<pattern>
  <id>latest</id>
  <size>1</size>
  <timestamp>any</timestamp>
  <car>any</car>
  <state>any</state>
  <cur_loc>any</cur_loc>
  <prev_loc>any</prev_loc>
  <next_loc>any</next_loc>
</pattern>
```

(a) latest 通用模板

```
<pattern>
  <id>silver_suv</id>
  <size>2</size>
  <timestamp>any</timestamp>
  <car>Silver SUV</car>
  <state>any</state>
  <cur_loc>any</cur_loc>
  <prev_loc>any</prev_loc>
  <next_loc>any</next_loc>
</pattern>
```

(b) 车辆模板 (以 Silver SUV 为例)

图 5-1: 两类模板

latest 模板相关。这条约束所蕴含的常理是，如果一辆车处于静止状态，那么它不可能经过某个传感器并产生一条上下文。这条约束会检查 latest 模板中最新的上下文的车辆状态，如果是静止就认为检测到不一致性错误。约束 B 与具体车辆相关，每辆车都有一个对应的版本，图 5-2b 是以银色小车为例。这条约束背后的逻辑是，车辆是有速度限制的，不可能在很短的时间内经过不同的传感器。所以约束 B 会从同一辆车的车辆模板 (图 5-1b) 中任意取出两条不同的上下文，如果它们的到达地点不同，但时间戳很近，就认为它们存在不一致性错误。与约束 B 一样，每辆车同样都有一条约束 C(图 5-2c) 的对应版本。约束 C 认为，根据物理空间的限制，车辆不可能是非连续的跳跃式前进，而必须先到达第一段路，才能进入下一段路。这条约束首先从车辆模板中取出任意一条上下文，然后判断要么它是产生最早的，要么在同一辆车的车辆模板里必定存在另一条旧上下文是在它的上一个路段产生的。否则就发生了一致性错误。

本实验中，SimCity 实验平台的控制中心运行在一台台式机上，配置为 Intel Core i7-4770 处理器 @ 3.40GHz，8GB 内存，搭载 Windows 8.1 专业版 64 位系统，JDK 版本为 Oracle Java 8。由于 SimCity 实验平台是由上下文驱动 (Context-driven) 的，每当有新的上下文变化，系统都会迅速处理完毕，然后在剩余的空闲时间里处于等待状态。SimCity 控制中心在上述 PC 上运行时，CPU 占用率在 0~2% 浮动，内存占用 200MB 左右。

5.2 用户界面

为了实时展示设备连接状况、车辆运行状态以及一致性错误检测与修复情况，方便用户随时开关运行场景、启停车辆、发布打车任务，SimCity 实验平

$$\begin{aligned} \forall \gamma_1 \in latest \\ not \\ still_state(\gamma_1.state) \end{aligned}$$

(a) 约束 A: 如果一辆车处于静止状态, 而传感器检测到了它的经过, 表明传感器出错了。

$$\begin{aligned} \forall \gamma_1 \in silver_suv \\ not \\ \exists \gamma_2 \in silver_suv \\ and \\ not \\ same(\gamma_1.id, \gamma_2.id) \\ not \\ same_location(\gamma_1.cur_loc, \gamma_2.cur_loc) \\ short_time(\gamma_1.timestamp, \gamma_2.timestamp) \end{aligned}$$

(b) 约束 B: 一辆银色小车不可能同时被不同的传感器检测到。

$$\begin{aligned} \forall \gamma_1 \in silver_suv \\ or \\ not \\ \exists \gamma_2 \in silver_suv \\ earlier_than(\gamma_2.timestamp, \gamma_1.timestamp) \\ \exists \gamma_3 \in silver_suv \\ and \\ same_location(\gamma_1.prev_loc, \gamma_3.cur_loc) \\ same_location(\gamma_1.cur_loc, \gamma_3.next_loc) \end{aligned}$$

(c) 约束 C: 银色小车必须按照拓扑顺序进入街道。例如, 它不能跳过当前街道而直接进入下一个。

图 5-2: 三类一致性约束

台的 PC 端控制中心提供了一个用户界面。启动控制中心后, 首先弹出车辆选择界面(图 5-3a), 每个选项都对应一辆车的名称, 括号中显示了该车的蓝牙串口传输服务 URL。点击 Done 按钮后, 就进入设备连接界面(图 5-3b), 设备连接与控制模块开始工作, 尝试与 10 个 EV3 程序块以及选择的车辆建立连接。我们从界面可以直接观察所有设备的通断状态。当所有设备都连通后, 设备连接界面自动切换到用户界面(图 5-4)。

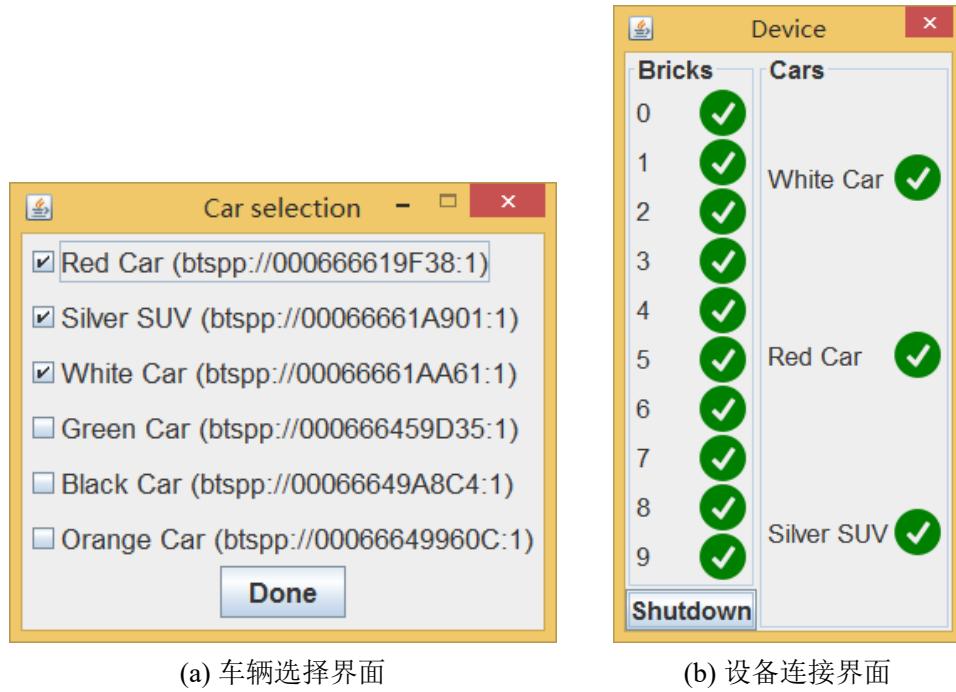


图 5-3: 启动界面

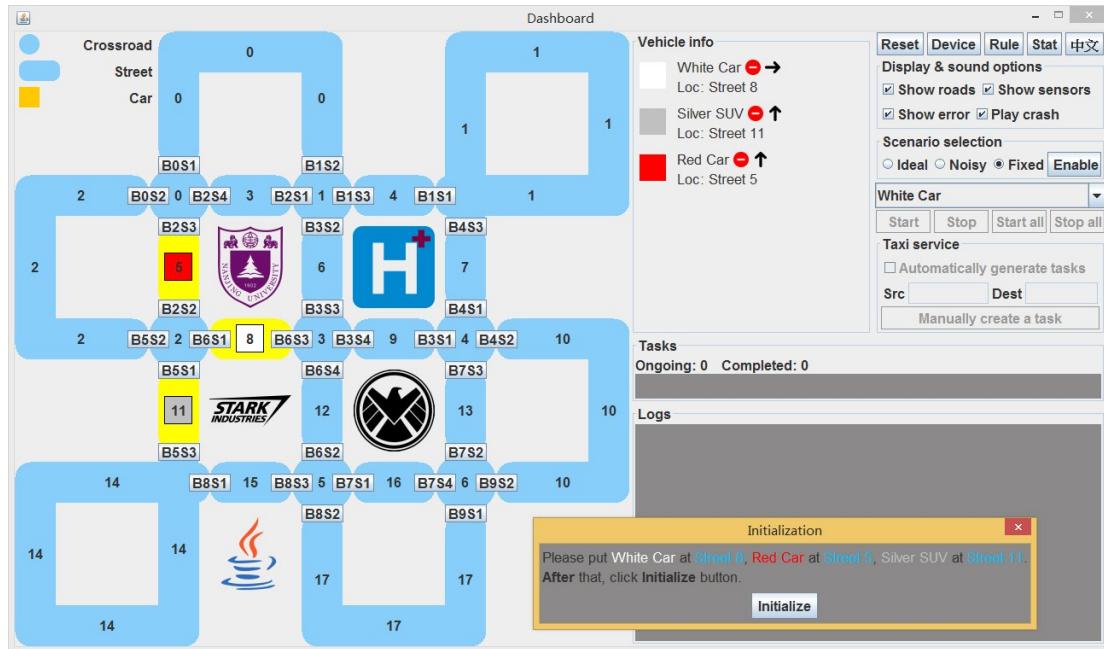
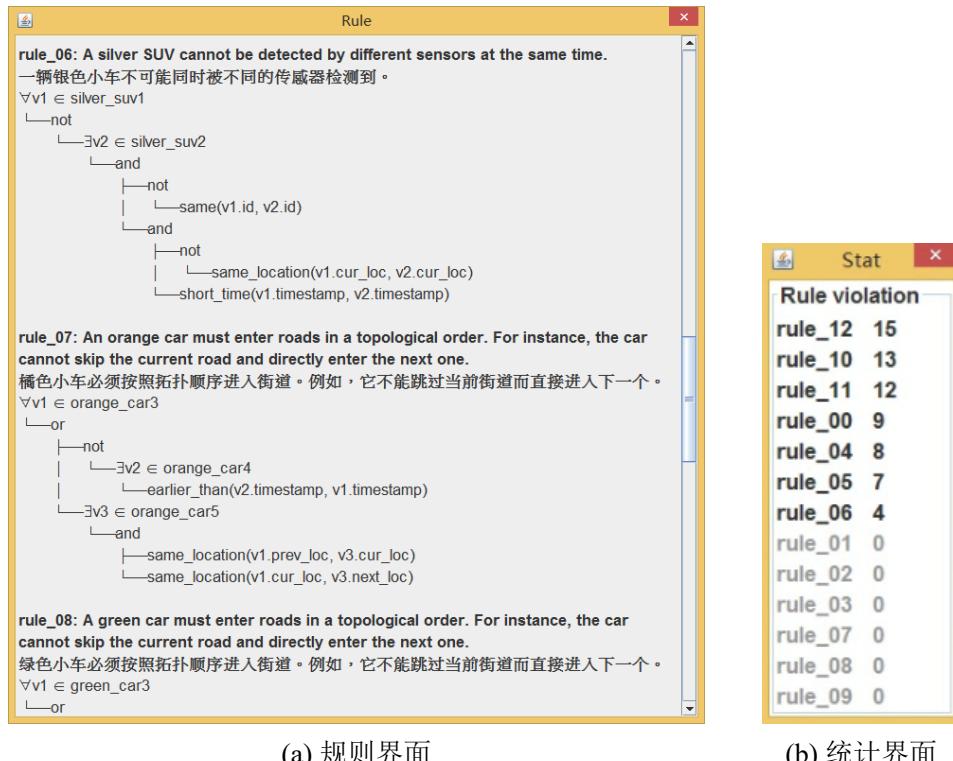


图 5-4: 用户界面

进入用户界面后，首先会弹出初始化对话框(图 5-4 右下角对话框)，要求用户遵照配置文件中编写的初始位置，将车辆放置在模拟城市的对应路段中，然后点击初始化 (Initialize) 按钮，将系统切换至重置状态，对车辆进行定位。



(a) 规则界面

(b) 统计界面

图 5-5: 规则界面与统计界面

定位完成后，系统切换回正常运行状态，我们就可以通过操作用户界面进行实验。不过在此之前，需要先讲解下用户界面的布局与功能。

用户界面的左侧是对应实景图(图3-5)的虚拟地图，左上角是图示。地图中的圆形路段表示十字路口(Crossroad)，圆角矩形表示单行道(Street)，城市空白处的5个图标代表5个虚拟建筑，从左到右，从上至下分别是：学校、医院、斯塔克工业、警察局、餐厅。这些建筑可以被选为打车应用的起始点或目的地。城市道路中的纯色方块代表在该路段上的车辆，方块颜色则对应车辆颜色。道路之间标有**SxBy**字样的按钮代表分隔路段的传感器编号。

在用户界面的右上部，从左到右分别有一个车辆信息区(Vehicle info)和一个操作区。在车辆信息区，所有选择的车辆的名称、地点会以文字显示，而图标、状态与方向会以图标显示。车辆信息区的右侧是操作区。在操作区的第一排有5个按钮，重置(Reset)按钮可以让用户手动地启动带车辆定位的重置功能，设备(Device)按钮可以查看图5-3b的设备连接界面，规则(Rule)按钮可以查看一致性检测与修复模块使用的所有一致性约束(图5-5a)，统计(Stat)按钮可以在开启修复错误的运行场景后查看各个规则被违反与修复的次数(图5-5b)，未使用的车辆所对应的规则永远会不被使用到，因此会自动被禁

用，点击中文按钮可以将用户界面的 UI 元素内容从英文切换到中文，同时中文按钮也切换为 English 按钮，再次点击可以切回中文。

一排按钮的下方是显示与声音选项区 (Display & sound options)，勾选显示路名 (Show roads) 与显示传感器 (Show sensors) 复选框，会在左侧虚拟地图上显示路段的编号以及传感器编号。在包含错误 (Noisy) 以及修复错误 (Fixed) 运行场景下，显示错误 (Show error) 与播放撞车声 (Play crash) 复选框可以被勾选，分别会产生在虚拟地图上显示检测或修复的一致性错误以及相撞车辆鸣笛示警的效果。

显示与声音选项区下方是运行场景选择区 (Scenario selection)，共有理想 (Ideal) 场景、包含错误 (Noisy) 的场景、修复错误 (Fixed) 的场景三种可以选择。未启用任何场景时系统处于正常运行的空闲状态。选择一个场景后，点击右侧的启用 (Enable) 按钮方可运行该场景，SimCity 系统进入正常工作，同时启用按钮变为关闭 (Disable) 按钮。点击关闭按钮会自动调用不带车辆定位的重置功能，对场景运行产生的数据进行清理。重置结束后场景成功关闭，系统再次进入空闲状态。

运行场景选择区下面是一个选择车辆的下拉列表，可以选中任意一个使用中的车辆。下拉列表下方是一排车辆操控按钮，点击启动 (Start) 按钮或停止 (Stop) 按钮会向自适应调度模块发送对下拉列表中选中车辆的前进或停止请求。点击全启动 (Start all) 或全停止 (Stop all) 按钮则会对所有使用车辆发送对应请求。对选中车辆发送前进/停止请求后，启动/停止按钮会被禁用，避免重复发送。而对所有车辆都发送了前进/停止请求时，全启动/全停止按钮同理也会被禁用。当运行场景关闭，系统处于空闲状态时，所有按钮都会被禁用。

车辆操控按钮的下方是打车服务区 (Taxi service)，用以发布打车任务。在运行场景未启用时，该区域同样是被禁用的。而在启用运行场景之后，用户既可以勾选自动产生任务 (Automatically generate tasks) 复选框，让打车应用自动产生 **使用车辆数 - 1** 个打车任务，并在一个任务完成时自动产生下一个，也可以点击手动创建任务 (Manually create a task) 按钮，进入任务创建模式。此时手动创建任务按钮会变为一个创建 (Create) 按钮和一个取消 (Cancel) 按钮，用户可以点击虚拟地图上的任意路段或建筑指定起点与终点，再点击创建按钮发布任务，或者点击取消按钮退出任务创建模式。在打车任务自动产生或手动创建后，打车应用会自动选择距离起始点最近的空闲车辆，将任务分配给它。

车辆信息区和操作区的下方是打车任务区 (Tasks)。这里会列出所有正在进

行的打车任务，并记录进行中 (Ongoing) 和已完成 (Completed) 的任务数量。再下面是记录区 (Logs)，与撞车事故、打车任务、错误修复相关的信息都会在记录区中滚动展示。

5.3 运行场景演示

本节会结合用户界面与实景俯视图，对 SimCity 实验平台的三种运行场景分别进行演示与说明。

5.3.1 理想场景 (Ideal Scenario)

启用理想场景后，一致性检测与修复模块作为中间件对上层模块透明。该模块默默地对设备连接与控制模块上交的上下文进行错误检测与修复后，只把修复后的一致上下文交给上层模块，而不提供任何的错误检测与修复信息。上层模块因此会认为这些上下文是由理想的传感器直接获得的，而不会在用户界面显示任何错误检测与修复信息。用户得以在理想场景中观察到，SimCity 实验平台在传感器不出错的条件下是怎样正常运行的。

如图 5-6a 所示，我们使用并连接上三辆车，分别是位于单行道 1 (Street 1) 的红车 (Red Car)、位于单行道 0 的银车 (Silver SUV) 和位于单行道 10 的白车 (White Car)。在启用理想场景，并勾选自动产生打车任务的选项后，在用户界面的打车任务区 (Tasks) 可以看到自动产生了两个打车任务：Clark 在十字路口 6 (Crossroad 6) 呼叫车辆，打车应用把他分配给最近的白车；Tony 在单行道 1 叫车，被分配给最近的银车。下面我们专注于 Tony 的打车任务。在点击全启动按钮后，三辆车都开始向前行驶。在图 5-6b 中，银车顺利到达 Tony 所在的单行道 1 后停下，开始载客上车。在右上部的车辆信息区中会发现，除了银车处于停止状态，红车也停下了。这是因为白车刚刚在十字路口 5 放下客人 Clark，正要离开，位于单行道 5 的红车只有停下等待白车离开后才能进入十字路口 5。银车在单行道 1 停止一段时间后打车应用认为乘客 Tony 已上车，于是会启动银车往目的地学校行进。在打车服务中，如果指定了建筑为起点或终点，车辆在行驶到建筑周围的路段时即视为抵达建筑。在图 5-6c 中，银车到达目的地学校周围的路段十字路口 1 后，认为已达终点，因此停下让乘客下车。此时位于单行道 0 的白车由于前方路段被银车占据，不得不停下等待。银车在等待一段时间后，认为乘客 Tony 已下车，因此继续前进等待下一个任务，如



图 5-6: 理想场景运行演示

图 5-6d。银车在离开十字路口 1 后，等待的白车终于被准许入内。

在理想场景下，所有车辆会在模拟城市中循环往复地前进、停止、相互避让，安全地完成一个个打车任务。用户可以从用户界面上实时观察到车辆的状态、方位以及打车任务的执行情况，也可以随时启动与停止车辆，但是车辆的最终行为还是由自适应调度模块决定。例如，如果白车前方有红车，此时即使用户选中白车并点击启动按钮，自适应调度模块也会停下白车，待红车离开后再令白车前进。

5.3.2 包含错误的场景 (Noisy Scenario)

在包含错误的场景下，一致性检测与修复模块只会对上下文进行错误检测，但不对一致性错误进行修复。它会把错误检测信息与原始的上下文分别交给用户界面与上层模块使用。因此我们可以在此场景下观察到 SimCity 实验平台在一致性错误的影响下是如何运行异常，调度失败，最终导致撞车事故的。

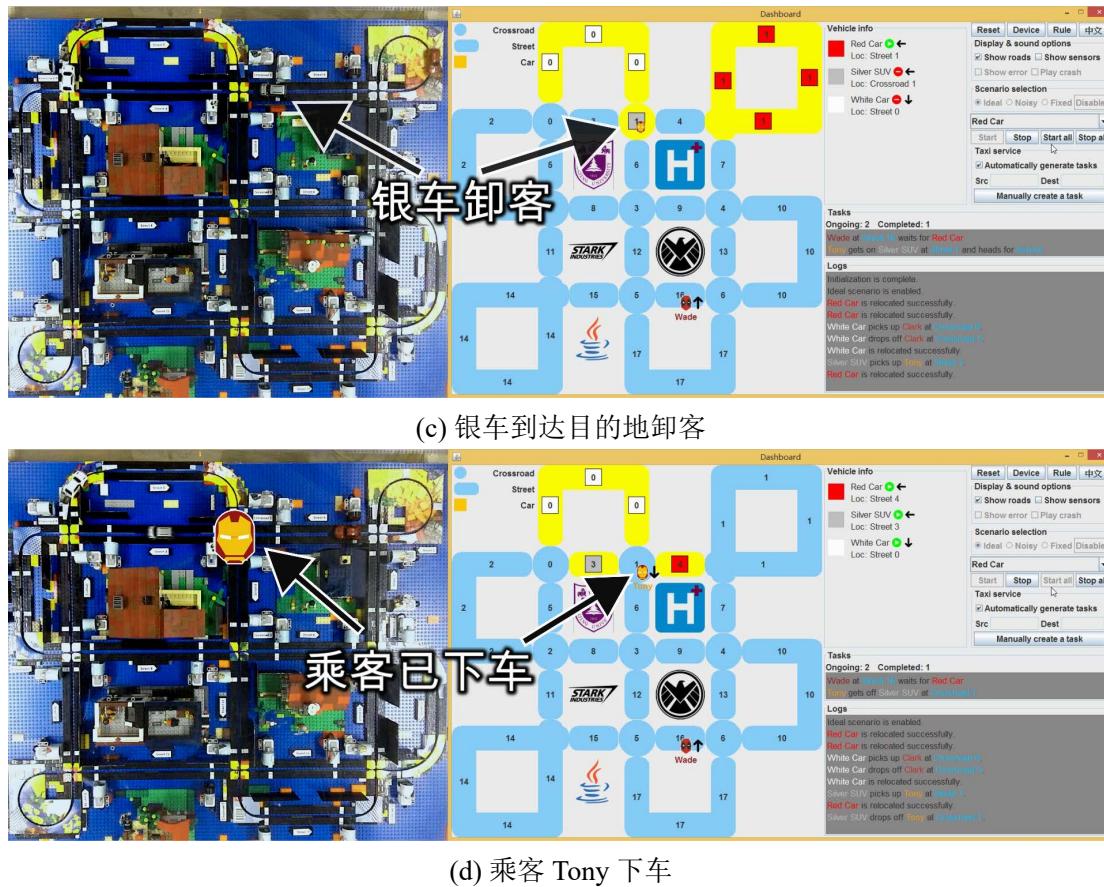


图 5-6: 理想场景运行演示 (续)

上层的用户界面会收到错误检测信息，此时在虚拟地图上会冒出粉红色气泡，表示对应位置的传感器采集到了一个错误的上下文数据。如图 5-7a 所示，传感器 B1S3 发生了误报 (False Positive, FP)，采集到一个银车经过的错误上下文。然而从同一时刻的图 5-7b 可以看出，此时银车位于单行道 1，根本没有经过传感器 B1S3。然而上层的事件监听模块、应用模块以及自适应调度模块会使用这个错误的上下文，它们会错误地认为银车位于十字路口 1 (银车错误位置图标为银色方块加白色问号)，而不是它的真实位置单行道 1 上 (银车真实位置图标为银色方块加白色叹号)。因此打车应用与车辆调度都会以银车的错误位置为准，而用户界面上的真实位置仅为展示之用。

包含错误的场景继续运行，在图 5-7c 中，当红车到达单行道 7 时，由于自适应调度模块使用了错误的上下文，误以为银车位于单行道 9，而红车前方的单行道 1 “无车”，也没有其他车辆准备进入，因此错误地允许红车进入单行道 1。红车继续前进，最终进入单行道 1 与银车发生了追尾事故 (图 5-7d)。



图 5-7: 包含错误的场景运行演示

SimCity 检测到撞车事故后，会强制停止所有车辆，并在记录区 (Logs) 中提示用户启动特定车辆以消除撞车事故，恢复场景的运行。

在包含错误的场景里，用户可以结合用户界面与物理环境，直观地观察到，在一致性错误检测与修复方法对错误检测或修复失败时，错误的上下文是怎样干扰应用模块的正常运行，并导致严重错误的。SimCity 实验平台通过使用错误的上下文做出错误反应，导致车辆的相撞，说明本实验平台能够找出一些对于一致性错误的检测不足或是不正确修复，具有一定的验证检测和修复正确性的能力。

5.3.3 修复错误的场景 (Fixed Scenario)

在修复错误的场景中，一致性检测与修复模块不仅会进行完整的错误检测与修复，还会将全部的检测与修复信息交给用户界面与上层模块。因此我们可以从用户界面上观察到对一致性错误的检测与修复是怎样时刻保障 SimCity 正



图 5-7: 包含错误的场景运行演示 (续)

常运行的。

用户界面收到一致性错误的修复信息后，会在虚拟地图上显示绿色气泡，如图 5-8a 所示。气泡表示对应位置的传感器采集到一个错误的上下文数据，但因为违反了一致性约束而被检测到并被成功修复。这里我们列举出两个错误修复并进行详细解释。在图 5-8b 中，传感器 B4S1 检测到银车经过，进入下面的单行道 7 中。但此时银车位于单行道 13，它不可能跳过中间的十字路口 4 而直接进入单行道 7。因为这个上下文数据与银车的以往上下文违反了规则 12，根据 Drop-latest 修复策略，我们认为最新的上下文是错误的，因此丢弃它，修复了传感器的错误。另一个错误修复的例子如图 5-8c 所示。银车位于单行道 7 并处于停止状态，但此时它前方的传感器 B4S3 却检测到它经过。根据规则 0，我们认为该传感器发生了误报，因此通过直接丢弃这个错误的上下文修复了一致性错误。

在运行修复错误的场景时，我们一方面观察到 SimCity 能够正常运行，调度多辆车顺利地执行打车任务；另一方面也在用户界面上看到陆续冒出的绿色

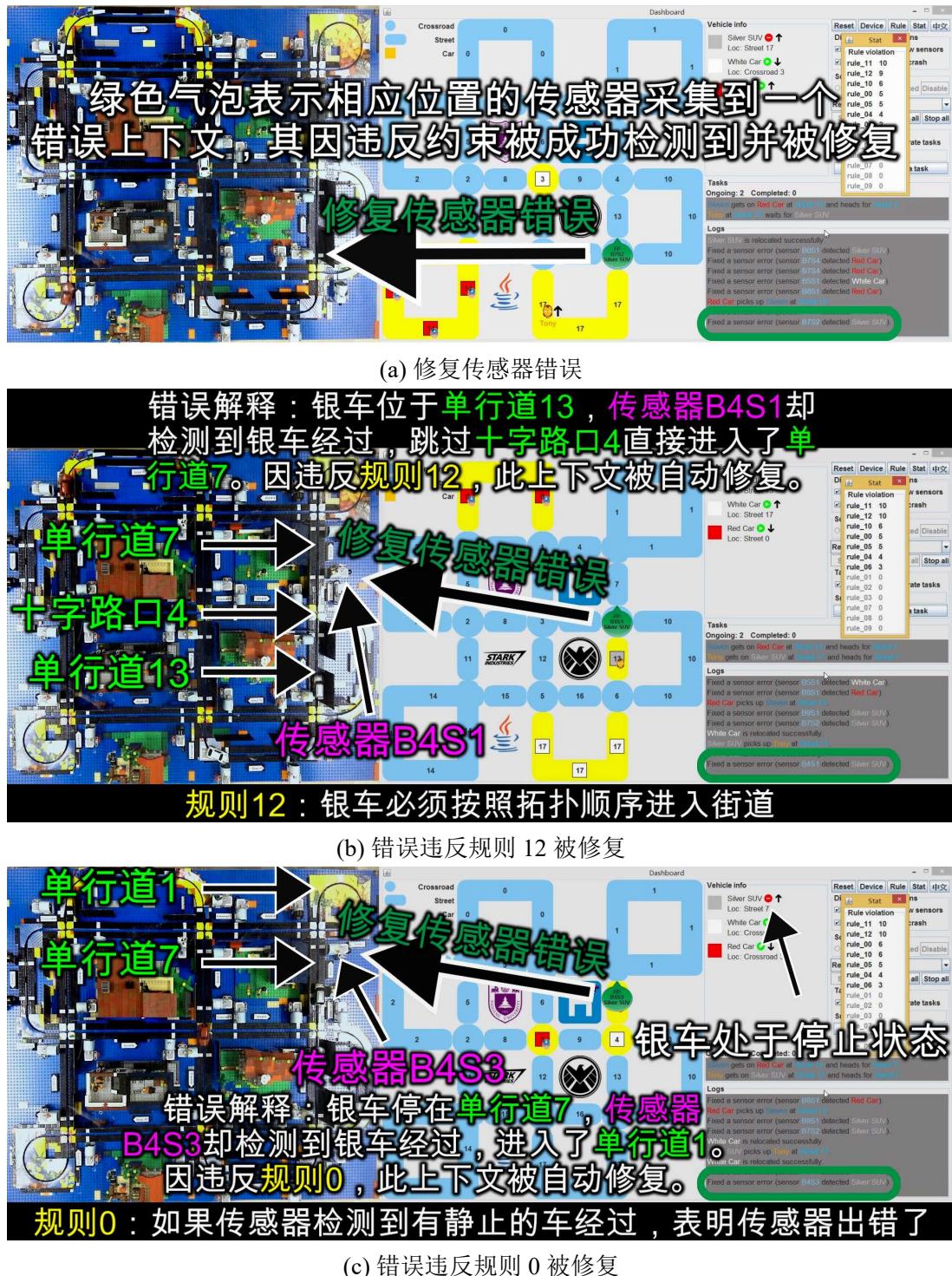


图 5-8: 修复错误的场景运行演示

气泡，表明作为中间件的一致性检测与修复模块正在后台不停地过滤不准确的上下文，保障上层的正常工作。一旦检测或修复发生错误，让不一致的上下文

被上层应用模块使用，SimCity 就会像运行包含错误的场景一样，因为对物理环境的错误感知做出错误反应，进而导致车辆相撞。

5.4 实验数据分析

我们使用了 1~3 辆车，让 SimCity 分别在包含错误的场景与修复错误的场景下连续运行，并对运行时间、处理上下文数据的总数及错误情况、一致性约束的违反情况、错误的修复情况、重定位情况、完成打车任务的数量以及自适应调度情况进行了统计。我们将分析这些实验数据来评估 SimCity 实验平台的性能。由于车辆电量的限制，我们不可能真正长时间地连续运行 SimCity 场景。当运行过程中某辆车因电量耗尽而自动关闭后，SimCity 会切换至暂停状态，此时我们会对所有车辆进行充电。所有车辆的电量充满后，我们再次打开车辆设备，让设备连接与控制模块重新连上车辆，从而恢复 SimCity 的运行。我们按照上述操作对车辆反复充电，分别让 SimCity 使用 1 辆车、2 辆车以及 3 辆车连续运行包含错误的场景与修复错误的场景，均持续 7200 秒，统计出本节中表格列出的实验数据。为了让所有车辆能时刻执行打车任务，我们将打车应用自动产生任务的数量提高为当前使用车辆数。

表 5-1: 包含错误的场景下的撞车事故

车辆数	运行时间 (秒)	检测到的 错误数量	撞车次数	运行时间/撞 车次数	错 误/撞 车次数
1	7200	764	N/A	N/A	N/A
2		695	79	91.1	8.8
3		770	181	39.8	4.3

我们首先依次使用 1~3 辆车连续运行包含错误的场景，只检测上下文中的一致性错误而不修复，并将发生的撞车事故的数据列在表 5-1 中。我们发现，检测到的一致性错误数量并未随使用车辆的增加而增长。我们认为，这是由于车辆增加的数量有限，并且车速较低，对于传感器而言物理环境的变化很小，因此出错的概率基本保持不变，产生的错误上下文所引起的不一致性错误的数量也就不会增长。观察撞车次数，只有一辆车行驶时不可能发生撞车，而当多辆车同时穿行在城市街道中，车辆在单行道上追逐或是在十字路口交汇的情况明显增多，撞车次数也随着车辆数直线上升。尤其是在使用 3 辆车时，平均每

运行 40 秒, SimCity 累计受到 4.3 个一致性错误的影响, 就会发生一次严重的撞车事故。这充分展现出上下文一致性错误对应用的危害性, 也证明了一致性错误处理对应用的正常运行是至关重要的。

表 5-2: 修复错误的场景下的上下文处理

车辆数	运行时间 (秒)	上下文总数	每秒产生上 下文数量	不一致上 下文数量	不一致上 下文占比
1	7200	8348	1.16	1219	14.6%
2		12202	1.69	1238	10.1%
3		15757	2.19	1086	6.9%

下一步, 我们同样使用了 1 至 3 辆车, 连续运行修复错误的场景。本段后面所有的表格数据都是在此场景下统计得到的。表 5-2 中列出了 SimCity 实验平台在此场景下对上下文数据的处理情况。可以看出, 在相同的运行时间内, SimCity 产生上下文的总数与速度并不与车辆数成正比。这是因为虽然车辆的增加会提高传感器产生上下文的数量, 但是车辆需要停下避让其他车辆的情况也增多了, 而在停止状态车辆就不会经过传感器产生新的上下文。因此上下文产生速度的增长会低于车辆数的增长。另一方面, 我们发现不一致上下文的数量不会随车辆数目发生变动, 其原因在前面已经解释过, 是传感器的出错率没有变动。于是我们看到, 随着车辆数目的提升, 虽然处理的上下文数据不断增加, 但是不一致上下文的数量基本不变, 导致它的占比也从 14.6% 降至 6.9%。

表 5-3: 一致性错误修复

车辆数	错误总数	漏报数	修复的错误数量	修复率	撞车次数
1	875	116	759	86.7%	N/A
2	884	139	745	84.3%	1
3	711	100	611	85.9%	3

表 5-3 给出了 SimCity 修复一致性错误的情况。在 §5.1 节中我们考虑到 SimCity 物理城市的环境特点, 采用 Drop-latest 修复策略对检测到的一致性错误进行修复。在实际运行中, 所有被检测出的错误都被正确地修复了。而对于由传感器漏报所引起的车辆重定位, 我们认为这也属于一致性错误, 但是 SimCity 的一致性检测与修复模块因为无法判断车辆位置, 无法正确地修复该

错误 (§4.8.3 节)。从表 5-3 中可以看出，传感器发生漏报错误的概率与车辆数无关，符合客观事实。将传感器的漏报错误视为未修复的一致性错误后，三次实验的错误修复率均达到 84% 以上。而由于漏报错误都交由重定位功能解决，SimCity 的上层模块不会收到任何不一致的上下文。

后两次实验中共发生了 4 起撞车事故，对比表 5-1 中高达 260 次的撞车次数，证明了一致性错误处理的有效性。我们分析了这几起撞车事故，发现都源自同一种情况：车辆经过**传感器 0** 时，由于**传感器 0** 所连接的 EV3 程序块的系统或无线网卡问题，导致车辆经过**传感器 0** 的上下文(记为**上下文 0**)迟迟未发送给 PC。车辆继续行驶到下一个**传感器 1**，**传感器 1** 所连接的另一个 EV3 程序块工作正常，将车辆经过**传感器 1** 的上下文(记为**上下文 1**)及时发送给 PC。此时虽然**上下文 1** 是准确的，但由于**上下文 0** 的缺失，约束 C(图 5-2c)被违反，Pcc 立即报告检测到了一致性错误。随后根据 Drop-latest 修复策略，**上下文 1** 被错误地丢弃。然后**上下文 0** 姗姗来迟，通过了约束检测后被加入到上下文集合中。SimCity 认为车辆经过了**传感器 0**，于是对通行时间 timeout 复位并重新开始计时。然而此时车辆已经驶过了**传感器 1**，又因为 timeout 复位而得以继续前进，最终追尾了前方的车辆。撞车事故的起因表面上看是**上下文 1** 被 Drop-latest 修复策略错误地丢弃，但我们通过分析发现这个所谓的一致性错误只是暂时出现，在缺失的**上下文 0** 到来后自然就会消失，也由此在 [12] 中被称为一致性错误冒险。因此事故产生的根本原因是 Pcc 检测方法的检测时机，在发生上下文变化时就立即进行了约束检测，导致了一致性错误的误报，令**上下文 1** 被误删。

如果采用构造模拟上下文或记录真实上下文的方式检验 Pcc，这种一致性错误冒险依然会被错误地检测出来。但是由于这两种检验方法脱离了真实的应用环境，无法将处理不当的上下文交给应用使用来造成应用的工作异常，因此难以发现冒险被错误地处理这一情况。而 SimCity 实验平台能够把经过一致性错误处理的上下文再交给上层模块使用，用户就可以通过用户界面与物理环境的异常现象迅速发现一致性错误被不当处理的情况，从而更有效地对一致性错误处理方法进行检验。

表 5-4 列出了一致性约束的违反情况。在三次实验中，约束 A(图 5-2a) 的检测次数都与表 5-2 中的上下文总数相同。这是因为约束 A 与通用上下文集合 latest 相关，产生的所有上下文都会一一加入到 latest 中，所以每产生一个上下文，latest 就变化一次，约束 A 就需要检测一次。在使用 1 辆车时，SimCity 只

表 5-4: 约束违反

车辆数	约束	检测次数	违反次数	违反率
1	约束 A(图 5-2a)	8348	87	1.04%
	约束 B(图 5-2b) 银车版本	8348	237	2.84%
	约束 C(图 5-2c) 银车版本	8348	605	7.25%
2	约束 A	12202	124	1.02%
	约束 B 黑车版本	6444	145	2.25%
	约束 B 白车版本	5758	118	2.05%
	约束 C 黑车版本	6444	286	4.44%
	约束 C 白车版本	5758	263	4.57%
3	约束 A	15757	132	0.84%
	约束 B 白车版本	5253	64	1.21%
	约束 B 红车版本	5099	51	1.00%
	约束 B 银车版本	5405	67	1.22%
	约束 C 白车版本	5253	189	3.60%
	约束 C 红车版本	5099	140	2.75%
	约束 C 银车版本	5405	205	3.79%

会采集到 1 辆车的上下文，因此约束 B(图 5-2b) 与约束 C(图 5-2c) 用到的车辆集合也和 latest 一样，会加入所有产生的上下文，所以约束 A 的检测次数等于产生的上下文总数。而在使用多辆车时，产生的上下文必然会加入其中一个车辆集合中，并且同一车辆的约束 B 与约束 C 所用车辆集合是相同的。因此约束 B 或约束 C 的检测次数之和必然也等同于上下文总数，且同一车辆的约束 B 与约束 C 的检测次数也相同。

再观察三次实验的规则违反率，都是约束 A 最低，约束 B 次之，约束 C 最高。因为违反约束 A 需要车辆在停止时恰好前方的传感器发生误报，再加上车辆大部分时间处于行驶状态，造成约束 A 违反率最低。而违反约束 B 要求在车辆经过一个传感器后的短时间内，前方的传感器恰好发生误报。这个条件也比较苛刻，但车辆在长时间的运行状态下都可能触发，因此违反率要高于约束 A。违反约束 C 的条件相对最简单，对车辆状态与时间均无要求，只需要车辆前方非最近的传感器发生误报即可，因此它也是最容易被检测出违反的。三次实验的另一个现象是，随着车辆的增多，各约束的违反率普遍降低。原因在分

析表 5-1 时已经做出解释，车辆增多，SimCity 产生的上下文也增多，一致性约束的检测次数会增长，但是传感器出错概率基本不变，因此违反约束的次数也不会同步上涨，最终导致违反率普遍下降。

表 5-5: 重定位

车辆数	重定位总数	自动重定位次数	成功率	漏报数	漏报占比
1	506	442	87.4%	116	22.9%
2	666	591	88.7%	139	20.9%
3	466	420	90.1%	100	21.5%

我们还记录了 SimCity 在运行过程中进行重定位的数据，列在表 5-5 中。在 §4.8.3 节我们解释过，在特定情况下一致性检测与修复模块无法判断是否发生了一致性错误，因而无法进行正确修复。重定位正是为解决这一特定情况的问题而设计的。我们发现重定位总数并没有随车辆的增加而增长，在 3 辆车的实验中甚至总数反而最少。我们认为这是车辆引发重定位的随机性较大所致。在 §3.4.2 节我们对车辆与路面分别进行了改装，由于车辆本身制造工艺和改装手艺的差别，不同车辆在不同路段上产生卡顿的程度也千差万别。而且在车辆数量增加以后，车辆会更频繁地停下进行避让。这进一步增加了车辆卡顿而引起重定位的随机性，因此重定位的总数会一定程度地波动。但是尽管有较大的随机性，我们的自动重定位算法的成功率依然达到 87% 以上。换句话说，平均每发生 10 次重定位，用户只需手动重定位车辆 1 次，证明了 SimCity 的高可用性。我们还统计了所有重定位中由传感器漏报引起的次数，占比为 20% 左右，表明传感器发生漏报的现象时有发生。目前我们虽然利用重定位解决了大部分的漏报错误，但未来有必要采取更好的检测与修复方法进行处理。

表 5-6: 打车任务与自适应调度

车辆数	完成任务总数	单个任务平均耗时(秒)	允许前进请求	拒绝前进请求	允许停止请求
1	170	42.4	4793	0	323
2	318	22.6	8141	627	637
3	482	14.9	12267	1011	987

最后，我们统计了打车应用与自适应调度模块的工作情况，列在表 5-6 中。完成任务的数量基本是与车辆数成比例上升的，因为我们让打车应用自动生成

与当前车辆数相同的任务，使得所有车辆都在不停地执行任务。当随机生成与完成的任务数量足够多，可以近似地认为一辆车完成一个任务所耗时间相同。那么使用两/三辆车好比双/三线程同时工作，在任务无穷尽的条件下，完成任务数会随车辆数等比例地增长。

在自适应调度模块的统计数据中，尤其值得关注的是拒绝前进请求这一项，它记录了自适应调度模块为避免相撞而停下车辆的次数，因此也可以视为成功阻止潜在的撞车事故的次数。当只有一辆车时，根本不存在碰撞的可能，因此模块拒绝它前进的次数为 0。而当运行多辆车时，撞车的风险明显提高，因此自适应调度模块拒绝前进请求从而成功阻止事故的次数也大幅度上涨。考虑到修复错误的场景中后两次实验共计四小时的运行时间内仅发生了四次撞车事故，这项数据证明了 SimCity 在使用一致上下文的前提下，有效调度车辆安全行驶的能力。

第六章 总结与展望

在本文中，我们设计并实现了一个结合硬件设备与软件系统的环境上下文一致性错误处理与展示平台 SimCity。我们搭建了一个物理的、遍布红外传感器的迷你城市，一方面通过蓝牙连接远程操控多个车辆在城市道路中行驶，另一方面不断地利用传感器采集与车辆相关的上下文数据。在对上下文进行一致性错误处理后，SimCity 系统中的上层模块会使用处理后的上下文，分派打车任务，并根据交通情况对车辆进行自适应调度，确保其顺利、安全地执行任务。而当一致性错误处理方法出错时，SimCity 系统上层使用了不一致的上下文，就会做出错误的反应，令车辆调度失败，最终酿成车祸。借此，SimCity 可以检验一致性检测与修复的正确性。

SimCity 实验平台具有完善的设备连接功能与状态转移功能，能够有效保障系统的正常运行。在系统启动时，SimCity 会自动地尝试与所有设备建立连接。在系统运行时，SimCity 会时刻检查设备的连通性，并在设备断开的第一时间暂停系统，尝试恢复连接。另外，我们统一设计各模块的工作线程框架，提供了安全地暂停模块并清除运行时数据的重置功能。为解决一致性错误处理无法判断传感器漏报的问题，我们又提出了后向重定位算法与递归的前向重定位算法。实验结果表明，自动重定位算法的成功率高达 87%。

我们为 SimCity 提供了一个可操作的用户界面与三种不同的运行场景，让用户可以操控车辆，并展现不同的实验效果。在理想场景中，我们可以查看 SimCity 正常工作的状况。在包含错误的场景中，我们可以观察一致性错误怎样一步步地误导系统的感知，影响它的行为，并最终做出错误的反应。在修复错误的场景中，我们可以看到一致性错误处理作为中间件有效地支撑着上层模块正确地运行，并可以实时观察一致性约束的违反与修复情况。

我们仔细设计了实验来分析 SimCity 实验平台的性能。我们使用了 1~3 辆车，分别在包含错误与修复错误的场景下连续运行两个小时，记录了上下文处理的实验数据。实验结果表明，SimCity 对一致性错误的正确修复率达到 84% 以上。我们使用 SimCity 对具有代表性的 Pcc 检测方法进行了检验。相比常用的构造模拟上下文与记录真实上下文的检验方法，我们额外地发现了 Pcc 对一

致性错误的误报，且该发现对所有在上下文产生变化时立刻进行检测的方法都适用。这证明 SimCity 比这两种检验方法拥有更好的检验效果。

目前本文的工作仍有一些不足有待改进。首先是 SimCity 的迷你城市能同时容纳与运行的车辆数目不多。在未来我们会进一步扩大城市规模，建立多层的复式结构。至于增加同时运行的车辆，由于蓝牙适配器有最大设备连接数的限制，我们考虑在一台 PC 上使用多个蓝牙适配器来同时遥控更多的车辆设备。另一个不足是处理传感器的漏报问题。目前 SimCity 的 Drop-latest 修复策略不能正确修复由漏报引起的一致性错误，而是使用重定位功能进行处理。未来我们希望利用机器学习探索漏报发生时的上下文模式，找到更有效的检测与修复方法解决这类错误。

参考文献

- [1] W Xi, C Xu, W Yang, and X Hong. How context inconsistency and its resolution impact context-aware applications. *Journal of Frontiers of Computer Science and Technology*, 8(4):427, 2014.
- [2] Chang Xu, Shing Chi Cheung, Wing Kwong Chan, and Chunyang Ye. Partial constraint checking for context consistency in pervasive computing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 19(3):9, 2010.
- [3] Chang Xu, Shing-Chi Cheung, and WK Chan. Incremental consistency checking for pervasive context. In *Proceedings of the 28th international conference on Software engineering*, pages 292–301. ACM, 2006.
- [4] Yingyi Bu, Jun Li, Shaxun Chen, Xianping Tao, and Jian Lv. An enhanced ontology based context model and fusion mechanism. *Embedded and Ubiquitous Computing–EUC 2005*, pages 920–929, 2005.
- [5] Yingyi Bu, Tao Gu, Xianping Tao, Jun Li, Shaxun Chen, and Jian Lu. Managing quality of context in pervasive computing. In *Quality Software, 2006. QSIC 2006. Sixth International Conference on*, pages 193–200. IEEE, 2006.
- [6] Jan Chomicki, Jorge Lobo, and Shamim Naqvi. Conflict resolution using logic programming. *IEEE Transactions on Knowledge and Data Engineering*, 15(1):244–249, 2003.
- [7] Insuk Park, Dongman Lee, and Soon J Hyun. A dynamic context-conflict management scheme for group-aware ubiquitous computing environments. In *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*, volume 1, pages 359–364. IEEE, 2005.
- [8] Chang Xu, Shing-Chi Cheung, Wing Kwong Chan, and Chunyang Ye. On impact-oriented automatic resolution of pervasive context inconsistency. In *Proceedings*

- of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 569–572. ACM, 2007.
- [9] Chang Xu, Shing-Chi Cheung, Wing Kwong Chan, and Chunyang Ye. Heuristics-based strategies for resolving context inconsistencies in pervasive computing applications. In *Distributed Computing Systems, 2008. ICDCS'08. The 28th International Conference on*, pages 713–721. IEEE, 2008.
- [10] 陈小康, 许畅, and 江磊. 非一致上下文的自动修复技术. *计算机科学与探索*, 7(4):326–336, 2013.
- [11] Chang Xu, Xiaoxing Ma, Chun Cao, and Jian Lu. Minimizing the side effect of context inconsistency resolution for ubiquitous computing. In *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, pages 285–297. Springer, 2011.
- [12] Wang Xi, Chang Xu, Wenhua Yang, Ping Yu, Xiaoxing Ma, and Jiang Lu. Shap: suppressing the detection of inconsistency hazards by pattern learning. In *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*, volume 1, pages 391–398. IEEE, 2014.
- [13] Wang Xi, Chang Xu, Wenhua Yang, Xiaoxing Ma, Ping Yu, and Jian Lu. Suppressing detection of inconsistency hazards with pattern learning. *Information and Software Technology*, 74:219–229, 2016.
- [14] Jun Sui, Chang Xu, Wang Xi, Yanyan Jiang, Chun Cao, Xiaoxin Ma, and Jian Lu. Gain: Gpu-based constraint checking for context consistency. In *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*, volume 1, pages 319–326. IEEE, 2014.
- [15] Anand Ranganathan, Jalal Al-Muhtadi, and Roy H Campbell. Reasoning about uncertain contexts in pervasive computing environments. *IEEE Pervasive computing*, 3(2):62–70, 2004.
- [16] Daniel Salber, Anind K Dey, and Gregory D Abowd. The context toolkit: aiding the development of context-enabled applications. In *Proceedings of the SIGCHI*

- conference on Human Factors in Computing Systems, pages 434–441. ACM, 1999.
- [17] Johannes Gehrke and Samuel Madden. Query processing in sensor networks. *IEEE Pervasive computing*, 3(1):46–55, 2004.
- [18] Christian Nentwich, Wolfgang Emmerich, Anthony Finkelsteiin, and Ernst Ellmer. Flexible consistency checking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(1):28–63, 2003.
- [19] Peri Tarr and Lori A Clarke. Consistency management for complex applications. In *Software Engineering, 1998. Proceedings of the 1998 International Conference on*, pages 230–239. IEEE, 1998.
- [20] Anand Ranganathan, Roy H Campbell, Arathi Ravi, and Anupama Mahajan. Conchat: A context-aware chat program. *IEEE Pervasive computing*, 1(3):51–57, 2002.
- [21] Anand Ranganathan and Roy H Campbell. An infrastructure for context-awareness based on first order logic. *Personal and Ubiquitous Computing*, 7(6):353–364, 2003.
- [22] Manuel Román, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H Campbell, and Klara Nahrstedt. A middleware infrastructure for active spaces. *IEEE pervasive computing*, 1(4):74–83, 2002.
- [23] Christian Nentwich, Licia Capra, Wolfgang Emmerich, and Anthony Finkelsteiin. xlinkit: A consistency checking and smart link generation service. *ACM Transactions on Internet Technology (TOIT)*, 2(2):151–185, 2002.
- [24] Chang Xu, YePang Liu, Shing Chi Cheung, Chun Cao, and Jian Lv. Towards context consistency by concurrent checking for internetwork applications. *Science China Information Sciences*, 56(8):1–20, 2013.
- [25] Chang Xu, Wang Xi, Shing-Chi Cheung, Xiaoxing Ma, Chun Cao, and Jian Lu. Cina: Suppressing the detection of unstable context inconsistency. *IEEE Transactions on Software Engineering*, 41(9):842–865, 2015.

-
- [26] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. Consistency management with repair actions. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 455–464. IEEE, 2003.
 - [27] Alexander Egyed. Fixing inconsistencies in uml design models. In *Proceedings of the 29th international conference on Software Engineering*, pages 292–301. IEEE Computer Society, 2007.
 - [28] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. *IEEE Transactions on software engineering*, 29(10):929–945, 2003.
 - [29] Chenhua Chen, Chunyang Ye, and Hans-Arno Jacobsen. Hybrid context inconsistency resolution for context-aware services. In *Pervasive Computing and Communications (PerCom), 2011 IEEE International Conference on*, pages 10–19. IEEE, 2011.
 - [30] Flaviu Cristian. Probabilistic clock synchronization. *Distributed computing*, 3(3):146–158, 1989.
 - [31] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
 - [32] Chang Xu and Shing-Chi Cheung. Inconsistency detection and resolution for context-aware middleware support. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 336–345. ACM, 2005.
 - [33] Colin J Fidge. Timestamps in message-passing systems that preserve the partial ordering. 1987.
 - [34] Friedemann Mattern et al. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.
 - [35] Alexander Egyed. Instant consistency checking for the uml. In *Proceedings of the 28th international conference on Software engineering*, pages 381–390. ACM, 2006.

简历与科研成果

基本信息

李晓帆，男，汉族，1992年3月26日出生，江苏扬州人。

教育背景

2014年9月—2017年6月 南京大学计算机科学与技术系 工程硕士
2010年9月—2014年6月 南京大学计算机科学与技术系 理学学士

攻读硕士学位期间完成的学术成果

1. 李晓帆, 许畅. 小车远程控制及自主寻路系统的设计与实现 [J]. 计算机科学, 2015, 42(12): 98-101.
2. 许畅, 马晓星, 吕建, 李晓帆. 一种上下文一致性检测与修复系统及检验方法与平台. 中国专利, 申请号/专利号: 201710177127.4. 申请日: 2017.03.23.

攻读硕士学位期间参与的科研课题

1. 国家重点基础研究发展计划（973课题）：持续演进的自适应网构软件模型、方法及服务质量保障（2015CB352202），马晓星（2015.01-2019.12）。
2. 国家自然科学基金（面上项目）：面向大数据环境的网构软件场景理解和功能及能耗保障研究（61472174），许畅（2015.01-2018.12）。

致 谢

时光荏苒，白驹过隙，在软件所的三年时光走得很慢，也很快。三年前，我实现本科毕业设计的场景还历历在目，一转眼，现在的我已经完成了硕士研究生的学业，即将步入社会。这充实快乐的三年时光让我深刻体会到自己在专业知识以及科研素养上的成长，在这里我要谢谢所有教导我、鼓励我与陪伴我的人！

我要感谢我的导师许畅教授。许老师在考虑问题时思维缜密，对我的工作要求精益求精。在完成本文工作的前期过程中，由于硬件经验的不足，我走了许多弯路，浪费了不少人力物力。许老师及时提点我转换思路，另辟蹊径。在步入正轨以后，许老师频繁地与我沟通进展，并用严谨的逻辑思路帮助我分析问题的解决方案以及后续工作。许老师治学严谨、风趣和蔼的品格，值得我在步入社会工作后不断学习。

我还要感谢软件所的各位老师。吕建老师、马晓星老师、陶先平老师、余萍老师、曹春老师、徐峰老师、黄宇老师、胡昊老师他们对各自领域内知识的掌握与钻研共同培养了软件所开放、多元、严谨的学术氛围，让我在三年的科研工作中受益良多。

我要感谢顾天晓、蒋炎岩、杨文华、金昊、奚旺、眭俊学长在科研学习上给予我的指导与帮助，感谢李其玮、钟远坤、李鑫、张瑞青、武翔宇、赵泽林、王瑶菁等同学在学习上的交流和生活中的相伴。

最后我要特别感谢我的家人与女友，三年里你们在生活中给予了我最大的支持与鼓励。没有你们对我的付出与坚持，我不可能完成充满压力的研究生学业。在步入社会以后，我会坚持正直的秉性，奋力拼搏，尽我所能来报答你们！

学位论文出版授权书

本人完全同意《中国优秀博硕士学位论文全文数据库出版章程》（以下简称“章程”），愿意将本人的学位论文提交“中国学术期刊（光盘版）电子杂志社”在《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》中全文发表。《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》可以以电子、网络及其他数字媒体形式公开出版，并同意编入《中国知识资源总库》，在《中国博硕士学位论文评价数据库》中使用和在互联网上传播，同意按“章程”规定享受相关权益。

作者签名：_____
____年____月____日

论文题名	SimCity：环境上下文一致性错误处理与展示平台研究				
研究生学号	MF1433021	所在院系	计算机科学与技术系	学位年度	2014
论文级别	<input type="checkbox"/> 硕士 <input checked="" type="checkbox"/> 硕士专业学位 <input type="checkbox"/> 博士 <input type="checkbox"/> 博士专业学位 (请在方框内画勾)				
作者电话	15996270557		作者 Email	leslie.ido@gmail.com	
第一导师姓名	许畅 教授		导师电话		

论文涉密情况：

不保密

保密，保密期：____年____月____日至____年____月____日

注：请将该授权书填写后装订在学位论文最后一页（南大封面）。

